john disc filename fred

## Additional notes for disc versions

Disc based versions of cassette softare have reference numbers that are prefixed by a '1' - eg SOFT914 becomes SOFT1914.

Unless otherwise specified, the function of the disc based program is identical to the cassette program, with the exception of the loading instructions where the usual disc loader:

```
RUN"DISC
```

should be used.

By examining the catalogue (directory) of the disc, you will be able to identify the various files that constitute the complete program, and whilst it may be possible to load and run these individual files, it is necessary to use the standard loader to ensure that the correct sequence of chaining is maintained.

Please note that discs are duplicated on both sides: generally the B side and A side are identical unless otherwise specified on the label or within the the supporting text.

# HiSoft PASCAL 4T

## Soft 155

Published by AMSOFT, a division of

Amstrad Consumer Electronics plc
Brentwood House
169 Kings Road
Brentwood
Essex
All rights reserved
First edition 1984

# SECTION 0 PRELIMINARIES

## 0.0 Introduction

Hisoft Pascal for the Amstrad CPC464 (HP464) is a fast, easy-to-use and powerful implementation of the Pascal language as specified in the Pascal User Manual and Report (Jensen/Wirth Second Edition).

There are a few omissions from this specification and these are as follows:

*FILE*s are not implemented although variables may be stored on tape.
A *RECORD* type may not have a *VARIANT* part.
*PROCEDURE*s and *FUNCTION*s are not valid as parameters.
In practice you should not find these omissions restrictive.

Many extra functions and procedures are included to reflect the changing environment in which compilers are used; among these are `POKE`, `PEEK`, `TIN`, `TOUT` and `ADDR`. Also, extra features have been added for the Amstrad CPC464 version to take advantage of the powerful facilities available with this computer - support for the event handling (`AFTER`, `EVERY` etc) is also provided.

In addition, a package of Turtle Graphics routines is included on Side 2 of the program cassette supplied. These Turtle Graphics routines are documented in Appendix 5.

Also supplied is a list of self-documenting routines for interfacing Hisoft Pascal to the CPC464 firmware, a list of these routines is given in Appendix 6.

The compiler occupies approximately 12K of storage while the runtimes take up roughly 5K and the editor 2K thus leaving you with some 20K for your programs and data. Hisoft Pascal runs as a foreground RAM program and takes over the CPC464 from its normal foreground program, BASIC. Thus it is not possible to mix BASIC and Hisoft Pascal, there should be no need for this since anything you can do in BASIC you can also do in Pascal.

To whet your appetite and to give you an immediate 'feel' for the package, we shall now present a short example of how to create, compile and run a Pascal program - remember though that there is no substitute for reading carefully through all sections of this manual and we urge you to do this before using the package for your own programs. In particular we recommend reading this section, the Editor section (Section 4) and working through the example programs given in Appendix 4.

## 0.0.1 Loading Instructions

Firstly, let's load the compiler into your machine: place the supplied cassette in the Datacorder with the label with 'Hisoft Pascal' on it facing up and press **PLAY** on the Datacorder. Now type: **CTRL [ENTER]** (hold **CTRL** and the small **[ENTER]** key down together)

A short BASIC loader program will be loaded and this will execute itself, produce a message and ask you:

```
RAM top ([ENTER] to default)?
```

At this point you should either enter a decimal number followed by **[ENTER]** or simply hit **[ENTER]** by itself. If you specify a number then this will be taken as the highest memory location that will NOT be used by the Pascal, all locations beneath this will be used and therefore corrupted. If you simply hit **[ENTER]** by itself in answer to the RAM top question then a default value of 45312 will be assumed: this gives the most room for your Pascal programs. In most cases hitting **[ENTER]** by itself will be sufficient; you only need specify a number if you wish to reserve some space at the top of memory for machine code programs that you want to interface to the Pascal.

Once you have specified a RAM top, or defaulted, then you will enter the internal editor supplied and a help screen will be displayed.

## 0.0.2 Example Program

Let us assume that you have defaulted (by pressing **[ENTER]**) the RAM top question load-up and you are now faced with the editor prompt '>' - enter `I` followed by **[ENTER]**.

The number `1Ø` should now appear at the left hand edge of the screen: this is a line number which the editor has automatically generated for you. The editor will continue to generate line numbers for you at the start of each line until you exit the `I` mode (see below). So, you are faced with this number `1Ø` - now type in the following program, ending each line by pressing the **[ENTER]** key and remembering that the line numbers are provided by the editor:

```
1Ø  program hanoi;
2Ø  var n : integer;
3Ø  procedure movedisk(sce,dest : integer);
 4Ø  begin
 5Ø  write(sce:1,' to ',dest:1,'. ')
6Ø  end;
7Ø  procedure move(n,sce,aux,dest : integer);
8Ø  begin
9Ø  if n=1 then movedisk(sce,dest)
1ØØ else
```

```
11Ø begin
12Ø move(n-1,sce,dest,aux);
13Ø movedisk(sce,dest);
14Ø move(n-1,aux,sce,dest)
15Ø end;
16Ø end;
17Ø (*MainBlock*)
18Ø begin
19Ø write('NumberofDiscs? ');
2ØØ read(n); writeln;
21Ø move(n,1,2,3)
22Ø end.
23Ø [ESC]
```

Note the use of a comment, enclosed by `(* *)` in line `17Ø` and remember that semi-colons '`;`' at the end of Pascal statements are important.

You should now be back in editor command mode, with that `>` prompt again. Now type the letter `C` followed by the **[ENTER]** key, this will compile your program into machine code and produce a compilation listing for you. At the end of the compilation, the message '`Run?`' should appear - if this does not happen and '`*ERROR*`' appears on the screen then press the letter **E** to get you back to the editor and then press the **[ENTER]** key. If you do get an error then check the program carefully against the above listing and retype any incorrect line: you can retype a line by typing its line number, followed by a space and then the body of the line, ending by hitting the **[ENTER]** key. Then compile again using `C`.

If there are no errors then respond to the '`Run?`' question with a letter `Y`. The program will now run and the first thing it does is ask you '`How many Discs?`'. The program is a Towers of Hanoi solution where you have three pegs with a number of different sized discs placed on one peg in descending order of size and the object is to move all the disks to another peg, maintaining the order. So now enter the number of disks you want placed on the first peg and press **[ENTER]**. Let's say you start off with three disks - so enter

**3 [ENTER]**

The program will now produce a listing of movements that will accomplish the movement of the discs from one peg to another. The '`Run?`' message will appear again when the program finishes: type '`Y`' or '`y`' to run the program again or any other key to return to the editor.

And that's all there is to it!! Now, please read the rest of the Sections in this manual carefully!

## 0.1 Scope of this manual.

This manual is not intended to teach you Pascal; you are referred to the separate Tutorial Guide to Hisoft Pascal and the excellent books given in the Bibliography if you are a newcomer to programming in Pascal.

This manual is a reference document, detailing the particular features of Hisoft Pascal.

Section 1 gives the syntax and the semantics expected by the compiler.

Section 2 details the various predefined identifiers that are available within Hisoft Pascal, from `CONST`ants to `FUNCTION`s.

Section 3 contains information on the various compiler options available and also on the format of comments.

Section 4 shows how to use the line editor which is an integral part of HP464.

The above Sections should be read carefully by all users.

Appendix 1 details the error messages generated both by the compiler and the runtimes.

Appendix 2 lists the predefined identifiers and reserved words.

Appendix 3 gives details on the internal representation of data within Hisoft Pascal - useful for programmers who wish to get their hands dirty.

Appendix 4 gives some example Pascal programs - study this if you experience any problems in writing Hisoft Pascal programs.

Appendix 5 contains details of the Turtle Graphics package supplied on Side 2 of the master tape.

Appendix 6 lists some useful Pascal procedures and functions to enable you to interface easily with the CPC464 firmware.

## 0.2 Compiling and Running.

For details of how to create, amend, compile and run an HP464 program using the integral line editor see Section 4 of this manual.

Once it has been invoked the compiler generates a listing of the form:

`xxxx nnnn` text of source line

where: `xxxx` is the address where the code generated by this line begins.
`nnnn` is the line number with leading zeroes suppressed.

If a line contains more than 80 characters then the compiler inserts new-line characters so that the length of a line is never more than 80 characters.

The listing may be directed to a printer, if required, by the use of compiler option `$P` (see Section 3).

You may pause the listing at any stage by hitting any key: subsequently use **[ESC]** to return to the editor or any other key to restart the listing.

If an error is detected during the compilation then the message '`*ERROR*`' will be displayed, followed by an up-arrow ('↑'), which points after the symbol which generated the error, and an error number (see Appendix 1). The listing will pause; hit '**E**' to return to the editor to edit the line displayed, '**P**' to return to the editor and edit the previous line (if it exists) or any other key to continue the compilation.

If the program terminates incorrectly (e.g. without '`END.`') then the message '`No more text`' will be displayed and control returned to the editor.

If the compiler runs out of table space then the message '`No Table Space`' will be displayed and control returned to the editor. You may allocate a different table size through use of the Editor's Alter command.

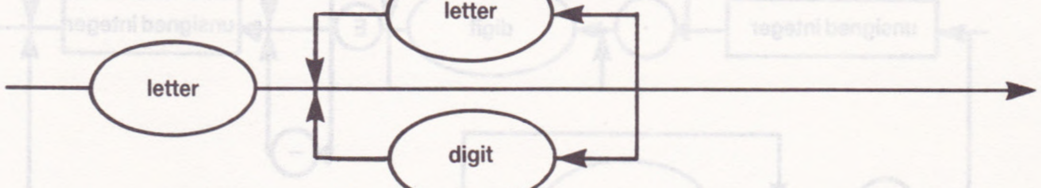
If the compilation terminates correctly but contained errors then the number of errors detected will be displayed and the object code deleted. If the compilation is successful then the message '`Run?`' will be displayed; if you desire an immediate run of the program then respond with '`Y`' or '`y`', otherwise control will be returned to the editor.

During a run of the object code various runtime error messages may be generated (see Appendix 1). You may suspend a run by hitting any key; subsequently hit **[ESC]** to abort the run or any other key to resume the run.

# SECTION 1 Syntax and Semantics

This section details the syntax and the semantics of Hisoft Pascal - unless otherwise stated the implementation is as specified in the Pascal User Manual and Report Second Edition (Jensen/Wirth).

## 1.1 IDENTIFIER.

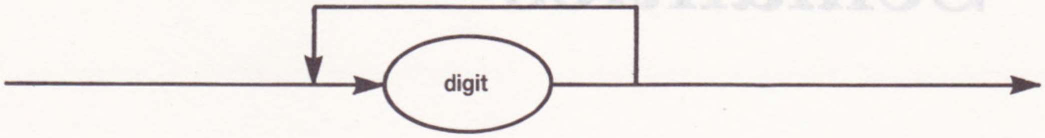


Only the first 10 characters of an identifier are treated as significant.

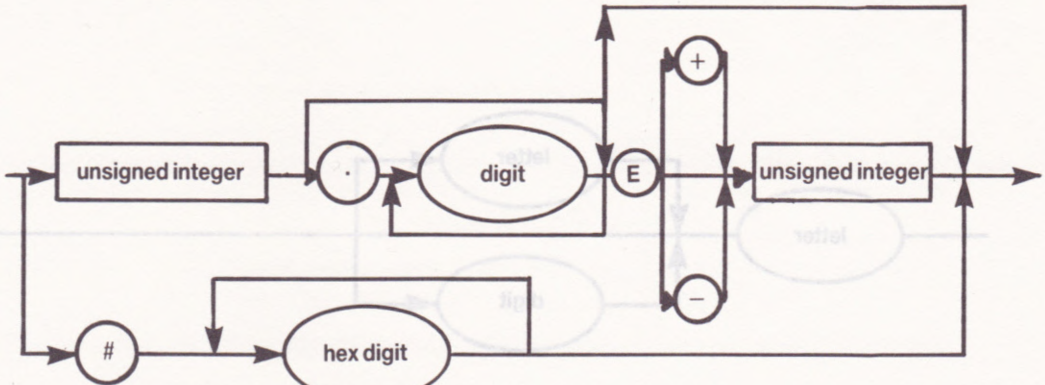
Identifiers may contain lower or upper case letters. Lower case is converted to upper case internally so that the identifiers `HELLO`, `HELLo` and `hello` are equivalent.

Reserved words and predefined identifiers may be entered in upper case or lower case and reserved words are converted to upper case by the editor.

## 1.2 Unsigned integer.



## 1.3 Unsigned number.



Integers have an absolute value less than or equal to 32767 in Hisoft Pascal. Larger whole numbers are treated as reals.

The mantissa of reals is 23 bits in length. The accuracy attained using reals is therefore about 7 significant figures. Note that accuracy is lost if the result of a calculation is much less than the absolute values of its arguments e.g. 2.00002 -2 does not yield 0.00002. This is due to the inaccuracy involved in representing decimal fractions as binary fractions. It does not occur when integers of moderate size are represented as reals e.g. 200002 - 200000 = 2 exactly.

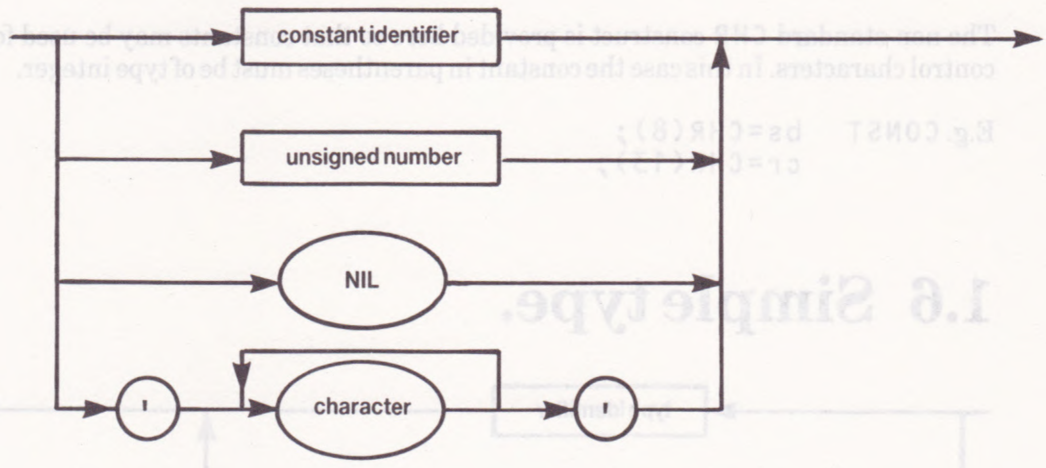
The largest real available is 3.4E38 while the smallest is 5.9E-39.

There is no point in using more than 7 digits in the mantissa when specifying reals since extra digits are ignored except for their place value.

When accuracy is important avoid leading zeroes since these count as one of the digits. Thus 0.000123456 is represented less accurately than 1.23456E-4.

Hexadecimal numbers are available for programmers to specify memory addresses for assembly language linkage inter alia. Note that there must be at least one hexadecimal digit present after the '#', otherwise an error (`*ERROR* 51`) will be generated.

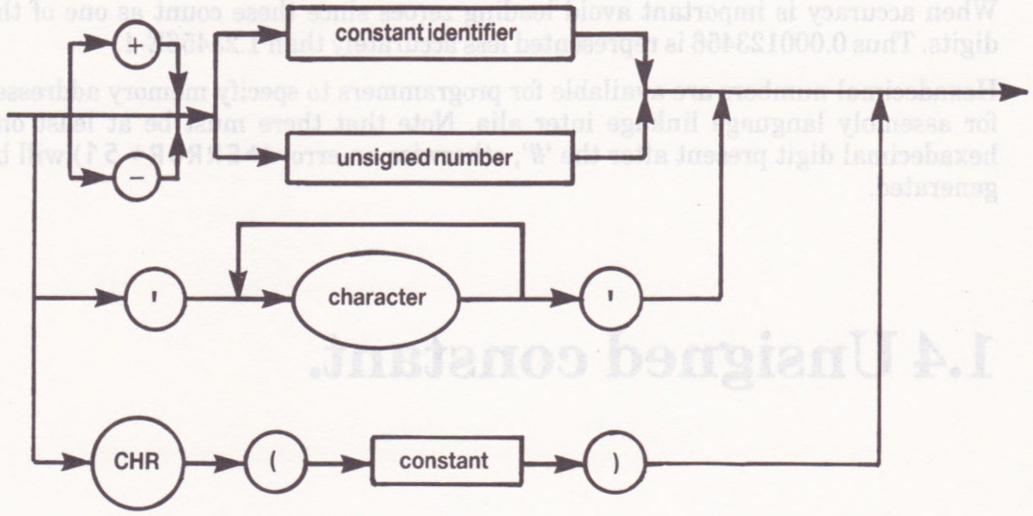
## 1.4 Unsigned constant.



Note that strings may not contain more than 255 characters. String types are `ARRAY [1..N] OF CHAR` where `N` is an integer between 1 and 255 inclusive. Literal strings should not contain end-of-line characters (`CHR(13)`) - if they do then an '`*ERROR* 68`' is generated.

The characters available are the full expanded set of ASCII values with 256 elements. To maintain compatibility with Standard Pascal the null character is not represented as `''''`; instead `CHR(Ø)` should be used.

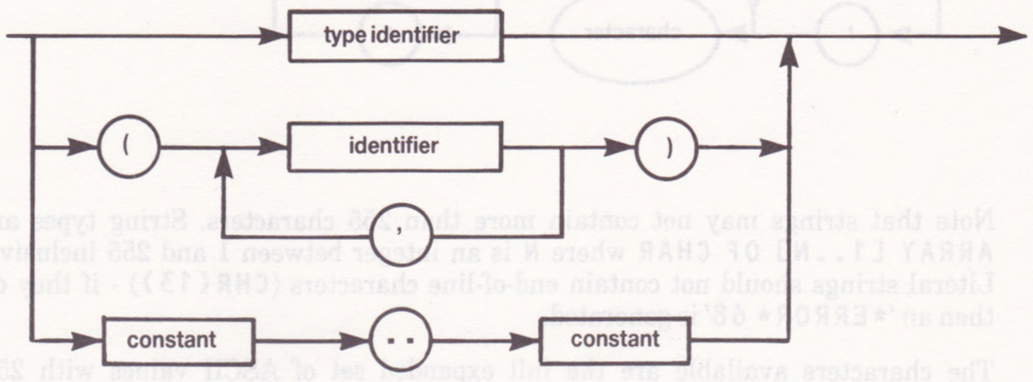
## 1.5 Constant.



The non-standard `CHR` construct is provided here so that constants may be used for control characters. In this case the constant in parentheses must be of type integer.

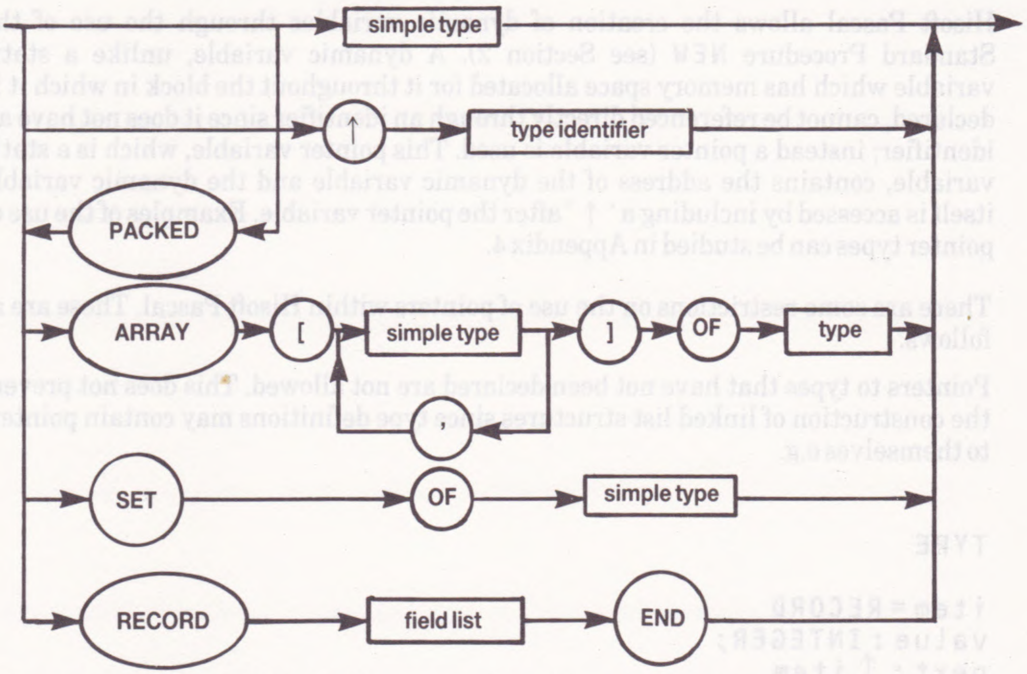
```
E.g. CONST   bs=CHR(8);
             cr=CHR(13);
```

## 1.6 Simple type.



Scalar enumerated types (identifier, identifier, ......) may not have more than 256 elements.

## 1.7 Type.



The reserved word `PACKED` is accepted but ignored since packing already takes place for arrays of characters etc. The only case in which the packing of arrays would be advantageous is with an array of Booleans - but this is more naturally expressed as a set when packing is required.

### 1.7.1 ARRAYs and SETs.

The base type of a set may have up to 256 elements. This enables SETs of CHAR to be declared together with SETs of any user enumerated type. Note, however, that only subranges of integers can be used as base types. All subsets of integers are treated as sets of 0..255.

Full arrays of arrays, arrays of sets, records of sets etc. are supported.

Two `ARRAY` types are only treated as equivalent if their definition stems from the same use of the reserved word `ARRAY`. Thus the following types are not equivalent:

```
TYPE

tablea=ARRAY[1..100] OF INTEGER;
tableb=ARRAY[1..100] OF INTEGER;
```

So a variable of type tablea may not be assigned to a variable of type tableb. This enables mistakes to be detected such as assigning two tables representing different data. The above restriction does not hold for the special case of arrays of a string type, since arrays of this type are always used to represent similar data.

## 1.7.2 Pointers.

Hisoft Pascal allows the creation of dynamic variables through the use of the Standard Procedure NEW (see Section 2). A dynamic variable, unlike a static variable which has memory space allocated for it throughout the block in which it is declared, cannot be referenced directly through an identifier since it does not have an identifier; instead a pointer variable is used. This pointer variable, which is a static variable, contains the address of the dynamic variable and the dynamic variable itself is accessed by including a ' ↑ ' after the pointer variable. Examples of the use of pointer types can be studied in Appendix 4.

There are some restrictions on the use of pointers within Hisoft Pascal. These are as follows:

Pointers to types that have not been declared are not allowed. This does not prevent the construction of linked list structures since type definitions may contain pointers to themselves e.g.

```
TYPE

item=RECORD
value:INTEGER;
next:↑item
END;
link=↑item;
```

Pointers to pointers are not allowed.

Pointers to the same type are regarded as equivalent e.g.

```
VAR
first:link;
current:↑item;
```

The variables first and current are equivalent (i.e. structural equivalence is used) and may be assigned to each other or compared.

The predefined constant NIL is supported and when this is assigned to a pointer variable then the pointer variable is deemed to contain no address.

## 1.7.4 RECORDs.

The implementation of RECORDs, structured variables composed of a fixed number of constituents called fields, within Hisoft Pascal is as Standard Pascal except that the variant part of the field list is not supported.

Two RECORD types are only treated as equivalent if their declaration stems from the same occurrence of the reserved word RECORD see Section 1.7.1 above.

Note that RECORD declarations do not open a new scope and as such you should not use the same field identifier in two different RECORD definitions at the same time.

e.g. If you have declared

```
rec1 = RECORD
  f1:integer
END;
```

then you should not use

```
rec2 = RECORD
  f1:integer
END;
```

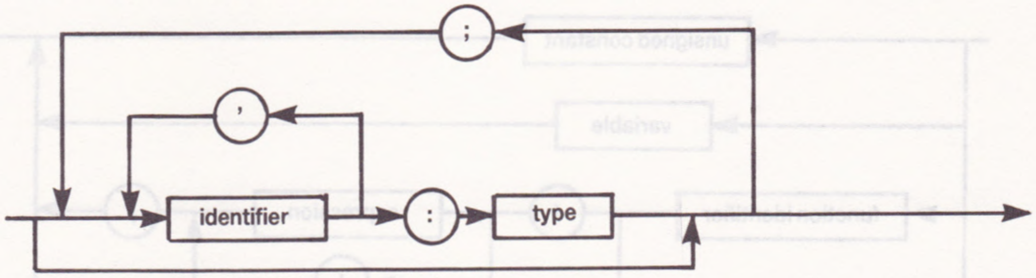
but, say,

```
rec2 = RECORD
  f2:integer
END;
```

The WITH statement may be used to access the different fields within a record in a more compact form. You should note that WITH statements cannot be called recursively and that WITH does not open a new scope.

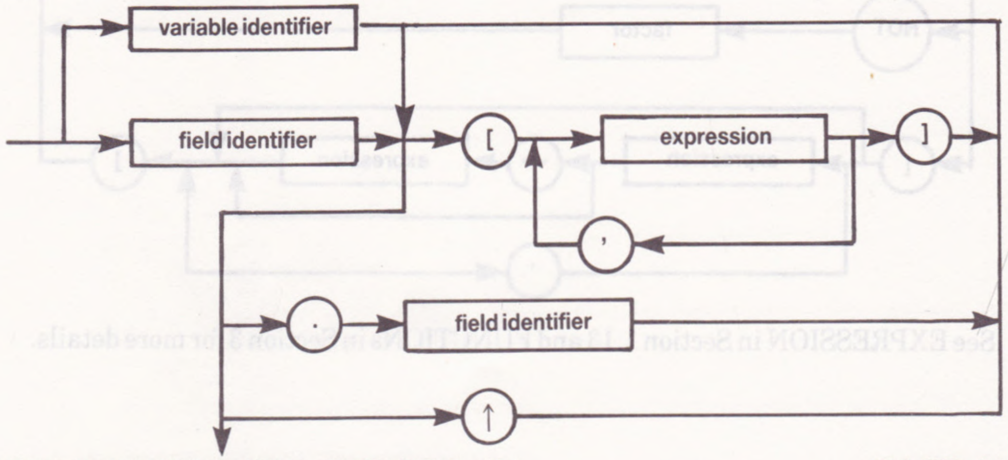
See Appendix 4 for an example of the use of WITH and RECORDs in general.

## 1.8 Field List.



Used in conjunction with RECORDs see Section 1.7.4 above and Appendix 4 for an example.

## 1.9 Variable.



Two kinds of variables are supported within Hisoft Pascal; static and dynamic variables. Static variables are explicitly declared through VAR and memory is allocated for them during the entire execution of the block in which they were declared.

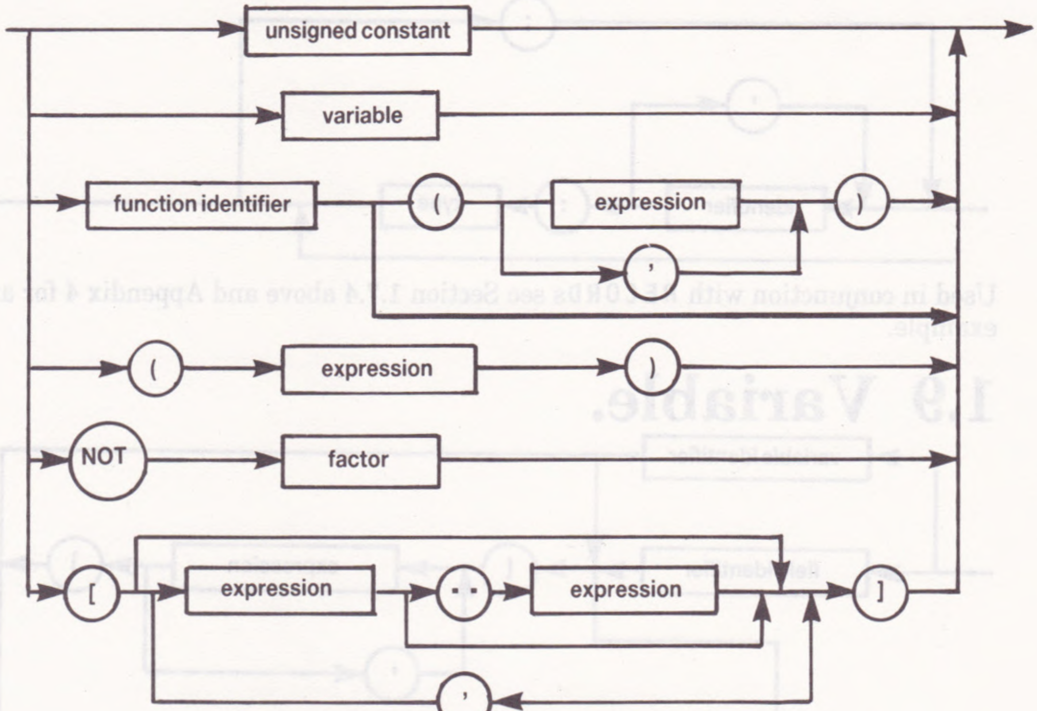
Dynamic variables, however, are created dynamically during program execution by the procedure NEW. They are not declared explicitly and cannot be referenced by an identifier. They are referenced indirectly by a static variable of type pointer, which contains the address of the dynamic variable.

See Section 1.7.2 and Section 2 for more details of the use of dynamic variables and Appendix 7 for an example.

When specifying elements of multi-dimensional arrays the programmer is not forced to use the same form of index specification in the reference as was used in the declaration.

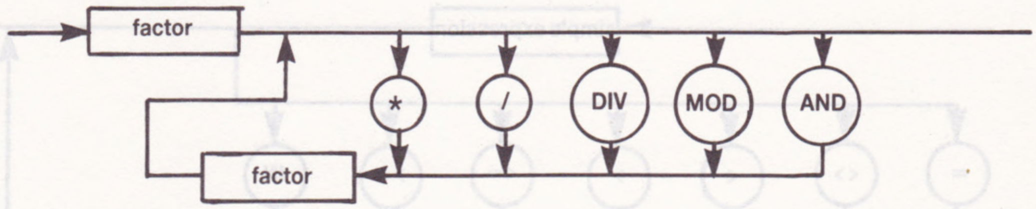
e.g. if variable a is declared as ARRAY[1..10] OF ARRAY[1..10] OF INTEGER then either a [1][1] or a [1,1] may be used to access element (1,1) of the array.

## 1.10 FACTOR



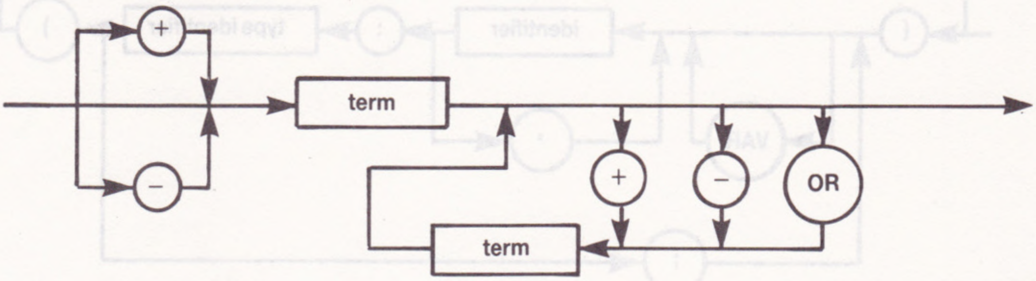
See EXPRESSION in Section 1.13 and FUNCTIONs in Section 3 for more details.

## 1.11 TERM.



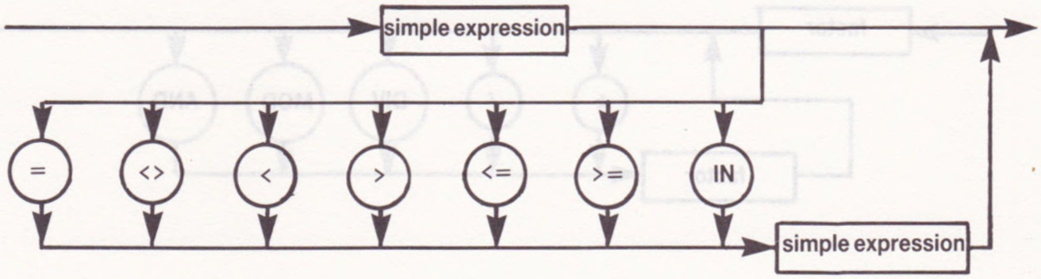
The lowerbound of a set is always zero and the set size is always the maximum of the maximum of the base type of the set. Thus a `SET OF CHAR` always occupies 32 bytes (a possible 256 elements - one bit for each element). Similarly a `SET OF Ø..1Ø` is equivalent to `SET OF Ø..255`

## 1.12 Simple Expression.



The same comments made in Section 1.11 concerning sets apply to simple expressions.

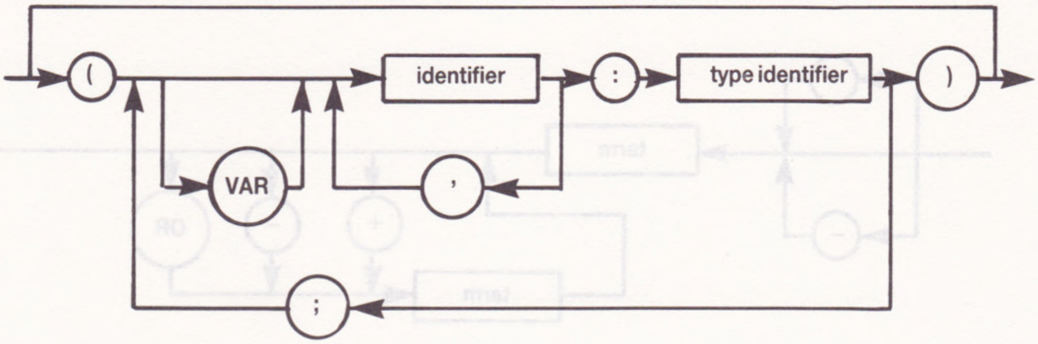
## 1.13 Expression.



When using `IN`, the set attributes are the full range of the type of the simple expression with the exception of integer arguments for which the attributes are taken as if [0..255] had been encountered.

The above syntax applies when comparing strings of the same length, pointers and all scalar types. Sets may be compared using `>=`, `<=`, `<>` or `=`. Pointers may only be compared using = and `<>`.

## 1.14 Parameter list.



A type identifier must be used following the colon - otherwise `*ERROR* 44` will result.

Variable parameters as well as value parameters are fully supported.

Procedures and functions are not valid as parameters.

## 1.15 Statement.

Refer to the syntax diagram on the following page.

***Assignment statements:***

See Section 1.7 for information on which assignment statements are illegal.

`CASE` ***statements:***

An entirely null case list is not allowed i.e. `CASE OF END;` will generate an error (`*ERROR* 13`).

The `ELSE` clause, which is an alternative to `END`, is executed if the selector ('expression' overleaf) is not found in one of the case lists ('constant' overleaf).

If the `END` terminator is used and the selector is not found then control is passed to the statement following the `END`.

`FOR` ***statements:***

The control variable of a `FOR` statement may only be an unstructured variable, not a parameter. This is half way between the Jensen/Wirth and ISO standard definitions.

`GOTO` ***statements:***

It is only possible to `GOTO` a label which is present in the same block as the `GOTO` statement and at the same level.

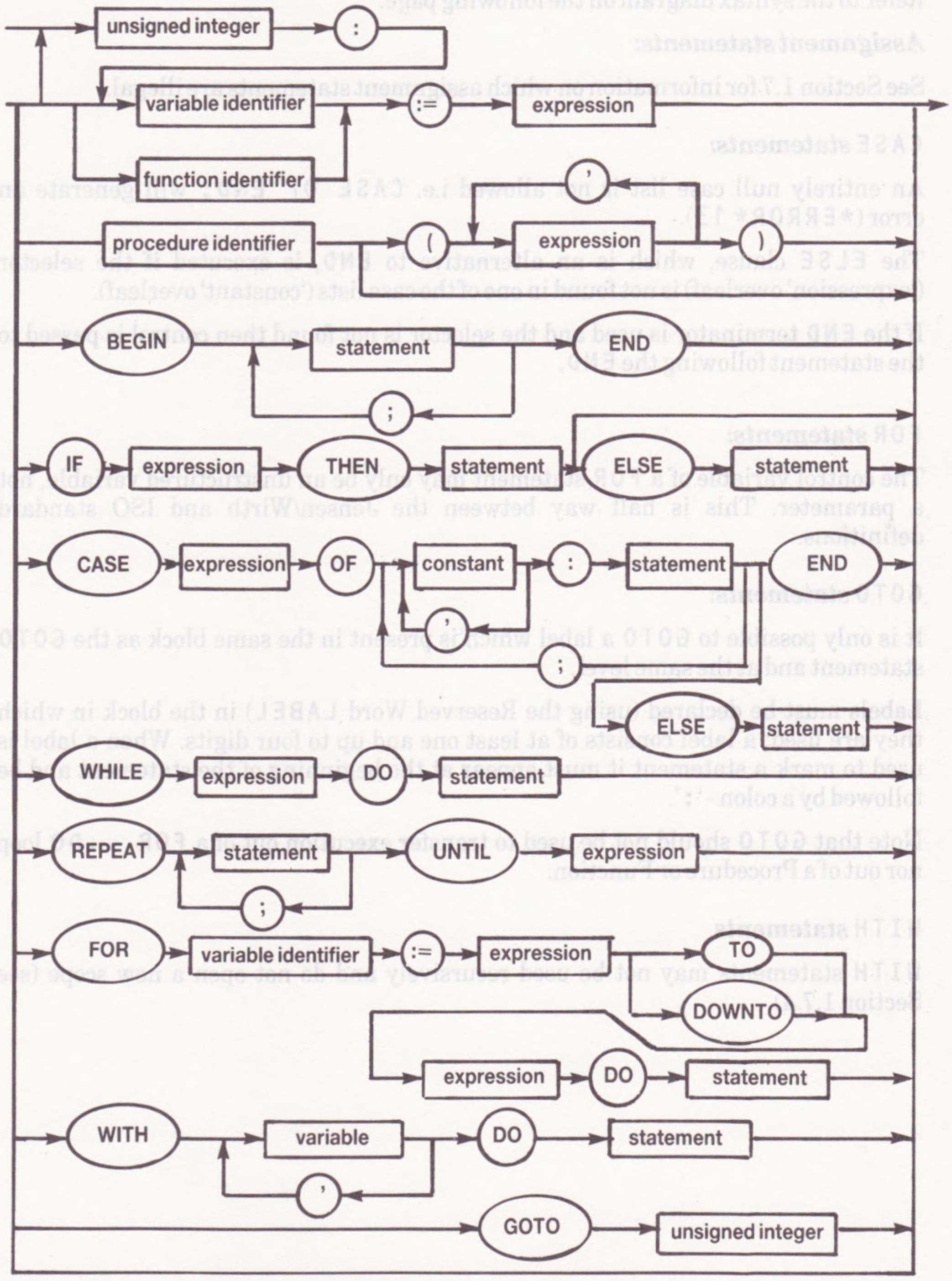
Labels must be declared (using the Reserved Word `LABEL`) in the block in which they are used; a label consists of at least one and up to four digits. When a label is used to mark a statement it must appear at the beginning of the statement and be followed by a colon - '`:`'.

Note that `GOTO` should not be used to transfer execution out of a `FOR...DO` loop nor out of a Procedure or Function.

`WITH` ***statements***

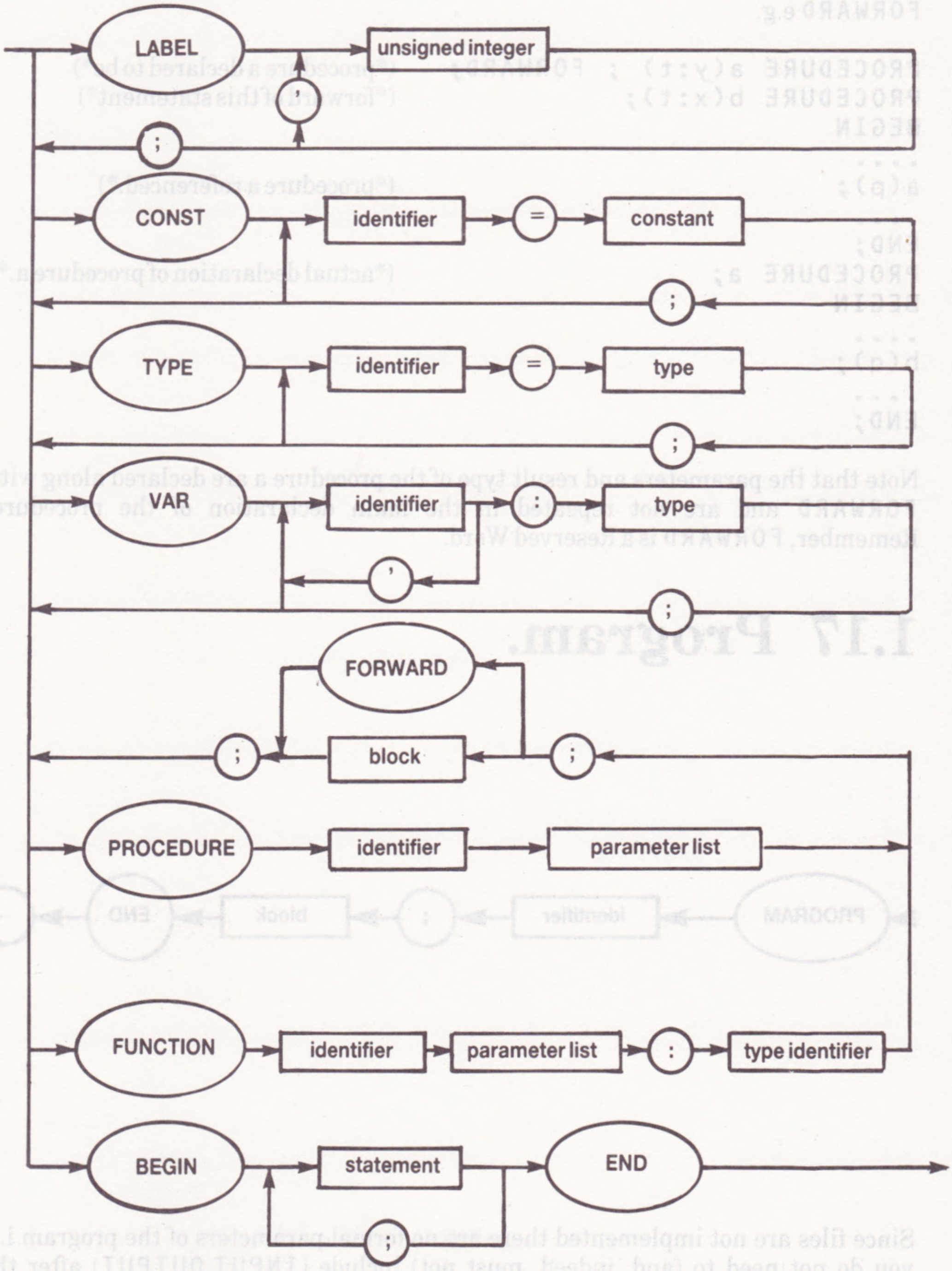
`WITH` statements may not be used recursively and do not open a new scope (see Section 1.7.4).

STATEMENT.



unsigned integer
:
variable identifier
:=
expression
function identifier
procedure identifier
(
expression
,
)
BEGIN
statement
END
;
IF
expression
THEN
statement
ELSE
statement
CASE
expression
OF
constant
:
statement
END
,
;
ELSE
statement
WHILE
expression
DO
statement
REPEAT
statement
UNTIL
expression
;
FOR
variable identifier
:=
expression
TO
DOWNTO
expression
DO
statement
WITH
variable
DO
statement
,
GOTO
unsigned integer

## 1.16 Block.


LABEL
unsigned integer
,
;
CONST
identifier
=
constant
;
TYPE
identifier
=
type
;
VAR
identifier
:
type
,
;
FORWARD
;
block
;
PROCEDURE
identifier
parameter list
FUNCTION
identifier
parameter list
:
type identifier
BEGIN
statement
;
END

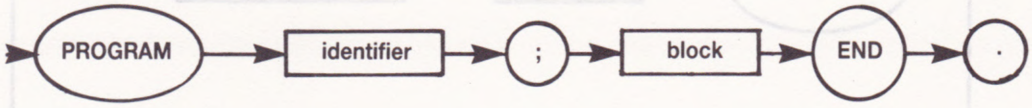
Forward References.

As in the Pascal User Manual and Report (Section 11.C.1) procedures and functions may be referenced before they declared through use of the Reserved Word FORWARD e.g.

```
PROCEDURE a(y:t) ; FORWARD;       (*procedure a declared to be*)
PROCEDURE b(x:t);                 (*forward of this statement*)
BEGIN
....
a(p);                             (*procedure a referenced.*)
....
END;
PROCEDURE a;                      (*actual declaration of procedure a.*)
BEGIN
....
b(q);
....
END;
```

Note that the parameters and result type of the procedure a are declared along with FORWARD and are not repeated in the main declaration of the procedure. Remember, FORWARD is a Reserved Word.

## 1.17 Program.



Since files are not implemented there are no formal parameters of the program i.e. you do not need to (and, indeed, must not) include (INPUT,OUTPUT) after the program name.

## 1.18 Strong TYPEing.

Different languages have different ways of ensuring that the user does not use an element of data in a manner which is inconsistent with its definition.

At one end of the scale there is machine code where no checks whatever are made on the TYPE of variable being referenced. Next we have a language like the Byte 'Tiny Pascal' in which character, integer and Boolean data may be freely mixed without generating errors. Further up the scale comes BASIC which distinguishes between numbers and strings and, sometimes, between integers and reals (perhaps using the '%' sign to denote integers). Then comes Pascal which goes as far as allowing distinct user-enumerated types. At the top of the scale (at present) is a language like ADA in which one can define different, incompatible numeric types.

There are basically two approaches used by Pascal implementations to strength of typing; structural equivalence or name equivalence. Hisoft Pascal uses name equivalence for RECORDs and ARRAYs. The consequences of this are clarified in Section 1 - let it suffice to give an example here; say two variables are defined as follows:

```
VAR  A : ARRAY['A'..'C'] OF INTEGER;
     B : ARRAY['A'..'C'] OF INTEGER;
```

then one might be tempted to think that one could write `A:=B;` but this would generate an error (`*ERROR* 10`) under Hisoft Pascal since two separate 'TYPE records' have been created by the above definitions. In other words, the user has not taken the decision that A and B should represent the same type of data. She/He could do this by:

```
VARA,B : ARRAY['A'..'C'] OF INTEGER;
```

and now the user can freely assign A to B and vice versa since only one 'TYPE record' has been created.

Although on the surface this name equivalence approach may seem a little complicated, in general it leads to fewer programming errors since it requires more initial thought from the programmer.

# SECTION 2 Predefined Identifiers.

## 2.1 Constants.

| | |
|---|---|
| `MAXINT` | The largest integer available i.e. 32767. |
| `FALSE, TRUE` | The constants of type Boolean. |

## 2.2 Types.

| | |
|---|---|
| `INTEGER` | See Section 1.3. |
| `REAL` | See Section 1.3. |
| `CHAR` | The full extended ASCII character set of 256 elements. |
| `BOOLEAN` | (`FALSE, TRUE`). This type is used in logical operations including the results of comparisons. |

## 2.3 Procedures and Functions.

### 2.3.1 Input and Output Procedures.

#### 2.3.1.1 WRITE

The procedure `WRITE` is used to output data to the screen or printer.

When the expression to be written is simply of type character then `WRITE(e)` passes the 8 bit value represented by the value of the expression `e` to the screen or printer as appropriate.

Note:

| | |
|---|---|
| `CHR(8)` | (**[CTRL]H**) gives a destructive backspace on the screen. |
| `CHR(12)` | (**[CTRL]L**) clears the screen or gives a new page on the printer. |
| `CHR(13)` | (**[CTRL]M**) performs a carriage return and line feed. |
| `CHR(16)` | (**[CTRL]P**) will normally direct output to the printer if the screen is in use or vice versa. |

Generally though:

`WRITE(P1,P2,.......Pn);` is equivalent to:
`BEGIN WRITE(P1); WRITE(P2); .......; WRITE(Pn) END;`

The write parameters P1,P2,......Pn can have one of the following forms:

`<e>` or `<e:m>` or `<e:m:n>` or `<e:m:H>`

where `e`, `m` and `n` are expressions and `H` is a literal constant.

We have 5 cases to examine:

1] `e` is of type integer: and either `<e>` or `<e:m>` is used.

The value of the integer expression `e` is converted to a character string with a trailing space. The length of the string can be increased (with leading spaces) by the use of `m` which specifies the total number of characters to be output. If `m` is not sufficient for `e` to be written or `m` is not present then `e` is written out in full, with a trailing space, and `m` is ignored. Note that, if `m` is specified to be the length of `e` without the trailing space then no trailing space will be output.

2] `e` is of type integer and the form `<e:m:H>` is used.

In this case `e` is output in hexadecimal. If `m=1` or `m=2` then the value (`e MOD 16` ↑ m) is output in a width of exactly `m` characters. If `m=3` or `m=4` then the full value of `e` is output in hexadecimal in a width of 4 characters. If `m>4` then leading spaces are inserted before the full hexadecimal value of `e` as necessary. Leading zeroes will be inserted where applicable. Examples:

```
WRITE(1Ø25:m:H);

m=1   outputs:  1
m=2   outputs:  Ø1
m=3   outputs:  Ø4Ø1
m=4   outputs:  Ø4Ø1
m=5   outputs: _Ø4Ø1
```

3] `e` is of type real. The forms `<e>`, `<e:m>` or `<e:m:n>` may be used.

The value of `e` is converted to a character string representing a real number. The format of the representation is determined by `n`.
If `n` is not present then the number is output in scientific notation, with a mantissa and an exponent. If the number is negative then a minus sign is output prior to the mantissa, otherwise a space is output. The number is always output to at least one decimal place up to a maximum of 5 decimal places and the exponent is always signed (either with a plus or minus sign). This means that the minimum width of the scientific representation is 8 characters; if the field width `m` is less than 8 then the full width of 12 characters will always be output. If `m>=8` then one or more decimal places will be output up to a maximum of 5 decimal places (`m=12`). For `m>12` leading spaces are inserted before the number. Examples:

```
WRITE(-1.23E10:m);

m=7   gives:  -1.23000E+10
m=8   gives:  -1.2E+10
m=9   gives:  -1.23E+10
m=10  gives:  -1.230E+10
m=11  gives:  -1.2300E+10
m=12  gives:  -1.23000E+10
m=13  gives:  _-1.23000E+10
```

If the form `<e:m:n>` is used then a fixed-point representation of the number `e` will be written with `n` specifying the number of decimal places to be output. No leading spaces will be output unless the field width `m` is sufficiently large. If `n` is zero then `e` is output as an integer. If `e` is too large to be output in the specified field width then it is output in scientific format with a field width of `m` (see above). Examples:

```
WRITE(1E2:6:2)        gives:  _00.00
WRITE(1E2:8:2)        gives:  __100.00
WRITE(23.455:6:1)     gives:  __23.5
WRITE(23.455:4:2)     gives:  _2.34550E+01
WRITE(23.455:4:0)     gives:  __23
```

4] `e` is of type character or type string.

Either `<e>` or `<e:m>` may be used and the character or string of characters will be output in a minimum field width of 1 (for characters) or the length of the string (for string types). Leading spaces are inserted if `m` is sufficiently large.

5] `e` is of type Boolean.

Either `<e>` or `<e:m>` may be used and 'TRUE' or 'FALSE' will be output depending on the Boolean value of `e`, using a minimum field width of 4 or 5 respectively.

## 2.3.1.2 WRITELN

`WRITELN` gives a newline. This is equivalent to `WRITE(CHR(13))` Note that a linefeed is included.

`WRITELN(P1,P2,........P3);` is equivalent to:
`BEGIN WRITE(P1,P2,.......P3); WRITELN END;`

## 2.3.1.3 PAGE

The procedure `PAGE` is equivalent to `WRITE(CHR(12));` and causes the video screen to be cleared or the printer to advance to the top of a new page.

## 2.3.1.4 READ

The procedure READ is used to access data from the keyboard. It does this through a buffer held within the runtimes - this buffer is initially empty (except for an end-of-line marker). We can consider that any accesses to this buffer take place through a text window over the buffer through which we can see one character at a time. If this text window is positioned over an end-of-line marker then before the read operation is terminated a new line of text will be read into the buffer from the keyboard. While reading in this line all the various control codes detailed in Section 0.0 will be recognised. Now:

READ(V1,V2,........Vn); is equivalent to:

BEGIN READ(V1); READ(V2);........; READ(Vn) END;

where V1, V2 etc. may be of type character, string, integer or real.

The statement READ(V); has different effects depending on the type of V. There are 4 cases to consider:

1] V is of type character.

In this case READ(V) simply reads a character from the input buffer and assigns it to V.

If the text window on the buffer is positioned on a line marker (a CHR(13) character) then the function EOLN will return the value TRUE and a new line of text is read in from the keyboard. When a read operation is subsequently performed then the text window will be positioned at the start of the new line.

**Important note:** Note that EOLN is TRUE at the start of the program. This means that if the first READ is of type character then a CHR(13) value will be returned followed by the reading in of a new line from the keyboard; a subsequent read of type character will return the first character from this new line, assuming it is not blank. See also the procedure READLN below.

2] V is of type string.

A string of characters may be read using READ and in this case a series of characters will be read until the number of characters defined by the string has been read or EOLN = TRUE. If the string is not filled by the read (i.e. if end-of-line is reached before the whole string has been assigned) then the end of the string is filled with null (CHR(Ø)) characters - this enables the programmer to evaluate the length of the string that was read.

The note concerning in 1] above also applies here.

3] V is of type integer.

In this case a series of characters which represent an integer as defined in Section 1.3 is read. All preceding blanks and end-of-line markers are skipped (this means that integers may be read immediately cf. the note in 1] above).

If the integer read has an absolute value greater than MAXINT (32767) then the runtime error 'Number too large' will be issued and execution terminated.

If the first character read, after spaces and end-of-line characters have been skipped, is not a digit or a sign ('+' or '−') then the runtime error 'Number expected' will be reported and the program aborted.

4] V is of type real.

Here, a series of characters representing a real number according to the syntax of Section 1.3 will be read.

All leading spaces and end-of-line markers are skipped and, as for integers above, the first character afterwards must be a digit or a sign. If the number read is too large or too small (see Section 1.3) then an 'Overflow' error will be reported, if 'E' is present without a following sign or digit then 'Exponent expected' error will be generated and if a decimal point is present without a subsequent digit then a 'Number expected' error will be given.

Reals, like integers, may be read immediately; see 1] and 3] above.

### 2.3.1.5 READLN

READLN(V1,V2,.......Vn); is equivalent to:
BEGIN READ(V1,V2,.......Vn); READLN END;

READLN simply reads in a new buffer from the keyboard; while typing in the buffer you may use the various control functions detailed in Section 0.0. Thus EOLN becomes FALSE after the execution of READLN unless the next line is blank.

READLN may be used to skip the blank line which is present at the beginning of the execution of the object code i.e. it has the effect of reading in a new buffer. This will be useful if you wish to read a component of type character at the beginning of a program but it is not necessary if you are reading an integer or a real (since end-of-line markers are skipped) or if you are reading characters from subsequent lines.

In general it is best to use a simple READLN after prompting for input and then follow this by calls to the procedure READ.

Example of READing variables of type CHAR:

```
PROGRAM READCHAR;
VAR CH:CHAR;
BEGIN
 REPEAT
  WRITE ('ENTER SOME CHARACTERS');
  READLN;
  WHILE NOT EOLN DO
   BEGIN
    READ (CH);
    WRITE ('The ASCII corresponding to 'CH,' is ',ORD(CH))
    END;
 UNTIL CH='E'
END.
```

## 2.3.2 Input Functions.

### 2.3.2.1 EOLN

The function `EOLN` is a Boolean function which returns the value `TRUE` if the next char to be read would be an end-of-line character (`CHR(13)`). Otherwise the function returns the value `FALSE`.

### 2.3.2.2 INCH

The function `INCH` causes the keyboard of the computer to be scanned and, if a key has been pressed, returns the character represented by the key pressed. If no key has been pressed then `CHR(Ø)` is returned. The function therefore returns a result of type character. This function should be used with the C- compiler option (see Section 3).

## 2.3.3 Transfer Functions.

### 2.3.3.1 TRUNC(X)

The parameter `X` must be of type real or integer and the value returned by `TRUNC` is the greatest integer less than or equal to `X` if `X` is positive or the least integer greater than or equal to `X` if `X` is negative. Examples:

`TRUNC(-1.5)` returns `-1`,   `TRUNC(1.9)` returns `1`

### 2.3.3.2 ROUND(X)

`X` must be of type real or integer and the function returns the 'nearest' integer to `X` (according to standard rounding rules). Examples:

| | |
|---|---|
| `ROUND(-6.5)` returns `-6` | `ROUND(11.7)` returns `12` |
| `ROUND(-6.51)` returns `-7` | `ROUND(23.5)` returns `24` |

### 2.3.3.3 ENTIER(X)

`X` must be of type real or integer - `ENTIER` returns the greatest integer less than or equal to `X`, for all `X`. Examples:

`ENTIER(-6.5)` returns `-7`   `ENTIER(11.7)` returns `11`

Note: `ENTIER` is not a Standard Pascal function but is the equivalent of BASIC's `INT`. It is useful when writing fast routines for many mathematical applications.

### 2.3.3.4 ORD(X)

X may be of any scalar type except real. The value returned is an integer representing the ordinal number of the value of X within the set defining the type of X.

If X is of type integer then ORD(X) = X ; this should normally be avoided.

Examples:

ORD('a') returns 97   ORD('@') returns 64

### 2.3.3.5 CHR(X)

X must be of type integer. CHR returns a character value corresponding to the ASCII value of X. Examples:

CHR(49) returns '1'   CHR(91) returns '['

## 2.3.4 Arithmetic Functions.

In all the functions within this sub-section the parameter X must be of type real or integer.

### 2.3.4.1 ABS(X)

Returns the absolute value of X (e.g. ABS(-4.5) gives 4.5). The result is of the same type as X.

### 2.3.4.2 SQR(X)

Returns the value X*X i.e. the square of X. The result is of the same type as X.

### 2.3.4.3 SQRT(X)

Returns the square root of X - the returned value is always of type real. A 'Maths Call Error' is generated if the argument X is negative.

### 2.3.4.4 FRAC(X)

Returns the fractional part of X: FRAC(X) = X - ENTIER(X).

As with ENTIER this function is useful for writing many fast mathematical routines. Examples:

FRAC(1.5) returns Ø.5   FRAC(-12.56) returns Ø.44

#### 2.3.4.5 SIN(X)

Returns the sine of X where X is in radians. The result is always of type real.

#### 2.3.4.6 COS(X)

Returns the cosine of X where X is in radians. The result is of type real.

#### 2.3.4.7 TAN(X)

Returns the tangent of X where X is in radians. The result is always of type real.

#### 2.3.4.8 ARCTAN(X)

Returns the angle, in radians, whose tangent is equal to the number X. The result is of type real.

#### 2.3.4.9 EXP(X)

Returns the value `e ↑ X` where `e = 2.71828`. The result is always of type real.

#### 2.3.4.10 LN(X)

Returns the natural logarithm (i.e. to the base `e`) of X. The result is of type real. If `X <= Ø` then a '`Maths Call Error`' will be generated.

## 2.3.5 Further Predefined Procedures.

#### 2.3.5.1 NEW(p)

The procedure `NEW(p)` allocates space for a dynamic variable. The variable `p` is a pointer variable and after `NEW(p)` has been executed `p` contains the address of the newly allocated dynamic variable. The type of the dynamic variable is the same as the type of the pointer variable `p` and this can be of any type.

To access the dynamic variable `p ↑` is used - see Appendix 4 for an example of the use of pointers to reference dynamic variables.

To re-allocate space used for dynamic variables use the procedures `MARK` and `RELEASE` (see below).

#### 2.3.5.2 MARK(v1)

This procedure saves the state of the dynamic variable heap to be saved in the pointer variable `v1`. The state of the heap may be restored to that when the procedure `MARK` was executed by using the procedure `RELEASE` (see below).

The type of variable to which `v1` points is irrelevant, since `v1` should only be used with `MARK` and `RELEASE`, never `NEW`.

For an example program using `MARK` and `RELEASE` see Appendix 4.

## 2.3.5.3. RELEASE(v1)

This procedure frees space on the heap for use of dynamic variables. The state of the heap is restored to its state when `MARK(v1)` was executed - thus effectively destroying all dynamic variables created since the execution of the `MARK` procedure. As such it should be used with great care.

See above and Appendix 4 for more details.

## 2.3.5.4 INLINE(C1,C2,C3,.........)

This procedure allows Z80 machine code to be inserted within the Pascal program; the values (`C1 MOD 256`, `C2 MOD 256`, `C3 MOD 256`) are entered in the object code starting from the current location counter address held by the compiler. `C1`, `C2`, `C3` etc. are integer constants of which there can be any number. Refer to Appendix 4 for an example of the use of `INLINE`.

## 2.3.5.5 USER(V)

`USER` is a procedure with one integer argument `V`. The procedure causes a call to be made to the memory address given by `V`. Since Hisoft Pascal holds integers in two's complement form (see Appendix 3) then in order to refer to addresses greater than `#7FFF (32767)` negative values of `V` must be used. For example `#9000` is `-28672` and so `USER(-28762);` would invoke a call to the memory address `#C000`. However, when using a constant to refer to a memory address, it is more convenient to use hexadecimal.

The routine called should finish with a Z80 RET instruction `(#C9)` and must preserve the `IX` register.

The registers A,B,C,D,E,H,L and F are set up from the values of RA,RB,RC,RD, RE,RH,RL and RF before calling the routine. The register values returned by the routine are given in the variables listed above. See section 2.4.2.

## 2.3.5.6 HALT

This procedure causes program execution to stop with the message '`Halt at PC=XXXX`' where `XXXX` is the hexadecimal memory address of the location where the `HALT` was issued. Together with a compilation listing, `HALT` may be used to determine which of two or more paths through a program are taken. This will normally be used during de-bugging.

## 2.3.5.7 POKE(X,V)

`POKE` stores the expression `V` in the computer's memory starting from the memory address `X`. `X` is of type integer and `V` can be of any type except `SET`. See Section 2.3.5.5 above for a discussion of the use of integers to represent memory addresses. Examples:

```
POKE(#6000,'A')        places #41 at location #6000.
POKE(-28672,3.6E3)     places 00 0B 80 70 (in hex.) at address #9000.
```

## 2.3.5.8 TOUT(NAME,START,SIZE)

TOUT is the procedure which is used to save variables on tape. The first parameter is of type ARRAY[1..12] OF CHAR and is the name of the file to be saved. SIZE bytes of memory are dumped starting at the address START. Both these parameters are of type INTEGER.

E.g. to save the variable V to tape under the name 'VAR' use:

```
TOUT('VAR_________',ADDR(V),SIZE(V))
```

The use of actual memory addresses gives the user far more flexiblity than just the ability to save arrays. For example all the global variables may be saved in one file. See Appendix 4 for an example of the use of TOUT.

## 2.3.5.9 TIN(NAME,START)

This procedure is used to load, from tape, variables etc. that have been saved using TOUT. NAME is of type ARRAY[1..12] OF CHAR and START is of type INTEGER. The tape is searched for a file called NAME which is then loaded at memory address START. The number of bytes to load is taken from the tape (saved on the tape by TOUT).

E.g. to load the variable saved in the example in Section 2.3.5.8 above use:

```
TIN('VAR_________',ADDR(V))
```

See Appendix 4 for an example of the use of TIN.

## 2.3.5.10 OUT(P,C)

This procedure is used to directly access the Z80's output ports without using the procedure INLINE. The value of the integer parameter P is loaded in to the BC register, the character parameter C is loaded in to the A register and the assembly instruction OUT (C),A is executed.

E.g. OUT(1,'A') outputs the character 'A' to the Z80 port 1.

## 2.3.5.11 EXTERNAL(S1, V1, V2,...)

This procedure allows calls to external ROMs and to RSXs in a way similar to BASIC's 'bar' command '|'. The first parameter is a string giving the name of the external command. This may then be followed by any number of parameters either of type integer, character or string. To pass a variable V to an external command use ADDR(V). Lower case in command names is converted to upper case. String descriptors are allocated on the heap.

Example: EXTERNAL('BASIC');
performs a return to BASIC in a controlled manner.

The main use of this procedure is to access the disk system. For example:

```
EXTERNAL('DIR','*.COM');
```

will give a directory of the .COM files on the current drive.

If you wish to use an RSX you should introduce it to the firmware using the firmware routine `KL LOG EXT` at the start of your program. This is neccessary because when a program finishes it clears all event queues which has the side-effect of clearing RSXs. See the CPC464 Firmware Manual for details concerning RSXs.

### 2.3.5.12 AFTER(COUNT,TIMER,PROC)

`AFTER` corresponds to the BASIC command of the same name. Its first parameter is an integer and is the count in 1/50ths of second after which the parameterless procedure which is its third parameter will be called. The second parameter is the timer to use. This should be an integer between `Ø` and `3` inclusive.

e.g `AFTER(1ØØ,1,FRED);` will cause the procedure `FRED` to be run after about 2 seconds, on timer 1.

### 2.3.5.13 EVERY(COUNT,TIMER,PROC)

`EVERY` corresponds to the BASIC command of the same name. Its parameters are the same as `EVERY` (See Section 2.3.5.12 above). See the BASIC manual for details of timers.

e.g. `EVERY (3ØØ,2,FRED);` will cause the Pascal procedure `FRED` to be invoked every 6 seconds. The procedure will be added to the ticker queue with a priority of 2.

### 2.3.5.14 SOUND (G, K, L, H, M, J, I)

The `SOUND` procedure has 7 integer parameters as follows:

1. Channels to use and rendezvous requirements (Channel status).
2. Amplitude envelope to use.
3. Tone envelope to use.
4. Tone period.
5. Noise period.
6. Initial amplitude.
7. Duration or envelope repeat count.

These correspond to the equivalent parameters of the BASIC SOUND command, but note that they are in a different order and that they must all be specified.

When a program terminates all sounds are stopped.

The sound program is allocated on the heap.

See Chapter 6, pages 6-8, of the BASIC user instructions for more detail.

## 2.3.5.15 ENV(N, P1, Q1, R1, P2, Q2, R2,...)

The `ENV` command corresponds to the BASIC command of the same name with integer parameters. Its first parameter is an envelope number (`1..15`). Then follow up to 5 envelope sections. Envelope sections may be either hardware or software controlled with each envelope section containing 3 integer parameters.

Software envelope sections have the following components:

1. Step count.
2. Step size.
3. Pause time.

See the BASIC manual for details.

Hardware sections have the following components:

1. Envelope shape (Greater than 128). The value of this parameter less 128 is sent to register 13 of the sound generator.
2. Low byte of envelope period which is sent to register 11.
3. High byte of envelope period which is sent to register 12.

## 2.3.5.16 ENT (S, T1, V1, W1, T2, V2, W2,....)

The `ENT` command corresponds to the BASIC command of the same name which has integer parameters. Its first parameter is an envelope number (`-15..15`). If the envelope number is negative then the envelope is a repeated one as in BASIC. Then follow up to 5 envelope sections. For relative sections the format is the same as in the BASIC command.

The absolute section corresponding to

```
    =toneperiod,pausetime     (2parameters)
```

in BASIC is

```
    240+toneperiod DIV 256, toneperiod MOD 256,
        pausetime      (3parameters)
```

in Hisoft Pascal. This is the same format as used by the firmware.

Refer to Chapter 8, pages 13-15 of BASIC User instructions for more detail of `ENV` and `ENT`.

## 2.3.6 Further Predefined Functions.

### 2.3.6.1 RANDOM(X)

RANDOM generates a pseudo-random number in the range Ø - MAXINT i.e. a positive INTEGER. RANDOM takes one parameter, if this parameter is zero then RANDOM(Ø) returns the next random number in the sequence otherwise the parameter is taken as the seed for a new random number sequence.

### 2.3.6.2 SUCC(X)

X may be of any scalar type except real and SUCC(X) returns the successor of X. Examples:

```
SUCC('A') returns 'B'          SUCC('5') returns '6'
```

### 2.3.6.3 PRED(X)

X may be of any scalar type except real; the result of the function is the predecessor of X. Examples:

```
PRED('j') returns 'i'      PRED(TRUE) returns FALSE
```

### 2.3.6.4 ODD(X)

X must be of type integer. ODD returns a Boolean result which is TRUE if X is odd and FALSE if X is even.

### 2.3.6.6 ADDR(V)

This function takes a variable identifier of any type as a parameter and returns an integer result which is the memory address of the variable identifier V. For information on how variables are held, at runtime, within Hisoft Pascal see Appendix 3. For an example of the use of ADDR see Appendix 4.

### 2.3.6.7 PEEK(X,T)

The first parameter of this function is of type integer and is used to specify a memory address (see Section 2.3.5.5). The second argument is a type; this is the result type of the function.

PEEK is used to retrieve data from the memory of the computer and the result may be of any type.

In all PEEK and POKE (the opposite of PEEK) operations data is moved in Hisoft Pascal's own internal representation detailed in Appendix 3. For example: if the memory from #9ØØØ onwards contains the values: 5Ø 61 73 63 61 6C (in hexadecimal) then:

WRITE(PEEK(#9ØØØ,ARRAY[1..6] OF CHAR)) gives 'Pascal'
WRITE(PEEK(#9ØØØ,CHAR)) gives 'P'
WRITE(PEEK(#9ØØØ,INTEGER)) gives 24912
WRITE(PEEK(#9ØØØ,REAL)) gives 2.46227E+29

see Appendix 3 for more details on the representation of types within Hisoft Pascal.

### 2.3.6.7 SIZE(V)

The parameter of this function is a variable. The integer result is the amount of storage taken up by that variable, in bytes.

### 2.3.6.8 INP(P)

`INP` is used to access the Z80's ports directly without using the procedure `INLINE`. The value of the integer parameter `P` is loaded into be `BC` register and the character result of the function is obtained by executing the assembly language instruction `IN A,(C)`.

### 2.3.6.9 INITEVENT(CLASS,PROC)

This function initialises an event block and returns its address as an integer. Its first parameter is the class of the event as detailed in the CPC 464 Firmware Manual. The event must be synchronous. The second parameter is a parameterless procedure which is called when the revelant event occurs. The event block is allocated on the heap. This function may be used by the advanced programmer to access the powerful features of the operating system's kernel that are not normally available in high level languages.

### 2.3.6.10 SQ(CHANNEL)

This function corresponds to the BASIC function `SQ` and returns an integer result giving the status of the channel corresponding to its integer parameter. See the BASIC manual for details. For example:

```
WRITE(SQ(1));
```

will give 4 if no `SOUND` commands have been issued for channel A.

### 2.3.6.11 REMAIN(TIMER)

`REMAIN` is a function with one integer parameter which corresponds to the BASIC function of the same name and returns an integer result giving the remaining count for the timer corresponding to its integer parameter. It also disables this timer.

## 2.4 Predefined variables

### 2.4.1 ERRFLG and ERRCHK

ERRFLG and ERRCHK are Boolean variables which may be used to trap erroneous user input when reading numbers. If ERRCHK is true then if a 'Digit Expected' error occurs, instead of halting the program ERRFLG is set to true and zero is read. Normally ERRFLG is false. For example

```
ERRCHK:=TRUE;
REPEAT
 READLN; READ(I)
 IF ERRFLG THEN WRITE('Please enter a number ')
UNTIL NOT ERRFLG;
```

### 2.4.2 RA,RB,RC,RD,RE,RH,RL and RF.

These variables of type char are used in conjunction with the USER procedure (See Section 2.3.5.5). Their values are used to initialise the Z80 registers before calling a USER routine and are updated with the corresponding values returned by the routine. For example:

```
RA:='a'; USER(#BB5A)
```

will output the character 'a' via the operating system routine TXT OUTPUT at address #BB5A.

After
```
USER(#BB06);
```

RA will contain the value returned by the operating system after calling the KM WAIT CHAR routine.
The predefined variables RAF,RBC,RDE and RHL are modified when assigning to RA,RB etc.

### 2.4.3. RAF,RBC,RDE and RHL.

These variables of type integer are used in conjunction with the USER procedure (See Section 2.3.5.5). Their values are used to initialise the Z80 registers before calling a USER routine and are updated with the corresponding values returned by the routine. For example:

```
RA:=CHR(I);
USER(#BCC5); {SOUND T ADDRESS}
IF ODD(RAF) THEN
      WRITE('Address of tone envelope',RHL:4:H)
 ELSE WRITE('Envelope not found')
```

will display the address of the envelope I if any. Note the use of ODD(RAF) which is true if the carry flag is set.

Assigning to RHL etc will modify the corresponding values of RH,RL etc. (See Section 2.4.2).

# SECTION 3 Comments and Compiler Options.

## 3.1 Comments.

A comment may occur between any two reserved words, numbers, identifiers or special symbols - see Appendix 2. A comment starts with a '{' character or the '(*' character pair. Unless the next character is a '$' all characters are ignored until the next '}'character or '*)' character pair. If a '$' was found then the compiler looks for a series of compiler options (see below) after which characters are skipped until a '}' or '*)' is found. The '{' and '}'characters may be obtained by using **SHIFT** and '[' together and **SHIFT** and ']' together respectively.

## 3.2 Compiler Options.

Compiler options are included in the program between comment brackets and the first option in the list is prefaced by a dollar symbol '$'.

Example:

`(*$C-,A-*)` to turn the keyboard and array checks off.

Option letters may be in upper or lower case.

The following options are available:

`Option L:`

Controls the listing of the program text and object code address by the compiler.

If `L+` then a full listing is given.
If `L-` then lines are only listed when an error is detected.

`DEFAULT: L+`

Option O:

Controls whether certain overflow checks are made. Integer multiply and divide and all real arithmetic operations are always checked for overflow.

If O+ then checks are made on integer addition and subtraction.
If O- then the above checks are not made.

DEFAULT: O+

Option C:

Controls whether or not keyboard checks are made during object code program execution. If C+ then pressing any key will pause the execution of the program, subsequently press **[ESC]** to abort the execution with a HALT (see section 2.3.5.6) or any other key to continue.

This check is made at the beginning of all loops, procedures and functions. Thus the user may use this facility to detect which loop etc. is not terminating correctly during the debugging process. It should certainly be disabled if you wish the object program to run quickly.
Use of C+ does not increase the size of the object program.

If C- then the above check is not made.

DEFAULT: C+

Option S:

Controls whether or not stack shecks are made.

If S+ then, at the beginning of each procedure and function call, a check is made to see if the stack will probably overflow in this block. If the runtime stack overflows the dynamic variable heap or the program then the message 'Out of RAM at PC=XXXX' is displayed and execution aborted. Naturally this is not foolproof; if a procedure has a large amount of stack usage within itself then the program may 'crash'. Alternatively, if a function contains very little stack usage while utilising recursion then it is possible for the function to be halted unnecessarily.

If S- then no stack checks are performed.
DEFAULT: S+

Option A:

Controls whether checks are made to ensure that array indices are within the bounds specified in the array's declaration.
If A+ and an array index is too high or too low then the message 'Index too high' or 'Index too low' will be displayed and the program execution halted.
If A- then no such checks are made.
DEFAULT: A+

Option I:

When using 16 bit 2's complement integer arithmetic, overflow occurs when performing a >, <, >=, or <= operation if the arguments differ by more than MAXINT (32767). If this occurs then the result of the comparison will be incorrect. This will not normally present any difficulties; however, should the user wish to compare such numbers, the use of I+ ensures that the results of the comparison will be correct. The equivalent situation may arise with real arithmetic in which case an overflow error will be issued if the arguments differ by more than approximately 3.4E38; this cannot be avoided.

If I- then no check for the result of the above comparisons is made.

DEFAULT: I-

Option P:

If the P option is used the device to which the compilation listing is sent is changed i.e. if the video screen was being used the printer is used and vice versa. Note that this option is not followed by a '+' or '-'.

DEFAULT: The video screen is used.

Option F:

This option letter must be followed by a space and then an twelve character filename. If the filename has less than twelve characters it must be padded with spaces.

The presence of this option causes inclusion of Pascal source text from the specified file from the end of the current line - useful if the programmer wishes to build up a 'library' of much-used procedures and functions on tape and then include them in particular programs.

The program should be saved using the built-in editor's 'P' command.

E.g.

```
(*$F MATRIX          include the text from a tape file MATRIX*)
```

When writing very large programs there may not be enough room in the computer's memory for the source and object code to be present at the same time. It is however possible to compile such programs by saving them to tape and using the 'F' option -then only 128 bytes of the source are in RAM at any one time, leaving much more room for the object code.

This option may not be nested.

The compiler options may be used selectively. Thus debugged sections of code may be speeded up and compacted by turning the relevant checks off whilst retaining checks on untested pieces of code.

# SECTION 4 The Integral Editor.

## 4.1 Introduction to the Editor.

The editor supplied with all versions of Hisoft Pascal is a simple, line-based editor designed to work with all Z80 operating systems while maintaining ease of use and the ability to edit programs quickly and efficiently. This editor has been enhanced for the AMSTRAD CPC464 by the addition of screen-editing facilities via use of the `COPY` key together with **SHIFT** cursors, these extra features are explained in detail below.

Text is held in memory in a compacted form; the number of leading spaces in a line is held as one character at the beginning of the line and all Reserved Words are tokenised into one character. This leads to a typical reduction in text size of 25%.

The editor is entered automatically when HiSoft Pascal is loaded from tape and displays the message:

```
HiSoft Pascal Amstrad CPC464
Version of 26/9/84
Copyright Hisoft 1983,4
All rights reserved
```

and then a help screen followed by the editor prompt '>'.

In response to the prompt you may enter a command line of the following format:

`C N1 , N2 , S1 , S2` followed by **[ENTER]**

where:

| | |
|---|---|
| `C` | is the command to be executed (see Section 4.2 below). |
| `N1` | is a number in the range 1-32767 inclusive. |
| `N2` | is a number in the range 1 -32767 inclusive. |
| `S1` | is a string of characters with a maximum length of 20. |
| `S2` | is a string of characters with a maximum length of 20. |

The comma is used to separate the various arguments (although this can be changed - see the '**Q**' command) and spaces are ignored, except within the strings. None of the arguments are mandatory although some of the commands (e.g. the '**D**'elete command) will not proceed without `N1` and `N2` being specified. The editor remembers the previous numbers and strings that you entered and uses these former values, where applicable, if you do not specify a particular argument within the command line.

The values of `N1` and `N2` are initially set to `10` and the strings are initially empty. If you enter an illegal command line such as `F-1,100,HELLO` then the line will be ignored and the message '`Pardon?`' displayed - you should then retype the line correctly e.g. `F1,100,HELLO`. This error message will also be displayed if the length of `S2` exceeds 20; if the length of `S1` is greater than 20 then any excess characters are ignored.

Commands may be entered in upper or lower case.

While entering a command line, certain control functions may be used e.g. **[CTRL]X** to delete to the beginning of the line, **[TAB]** to advance the cursor to the new tab position.

The following sub-section details the various commands available within the editor - note that wherever an argument is enclosed by the symbols '< >' then that argument 'must' be present for the command to proceed.

## 4.2 The Editor Commands.

### 4.2.1 Text Insertion.

Text may be inserted into the textfile either by typing a line number, a space and then the required text or by use of the 'I' command. Note that if you type a line number followed by **[ENTER]** (i.e. without any text) then that line will be deleted from the text if it exists. Whenever text is being entered then the control functions **[CTRL]X** (delete to the beginning of the line), **[TAB]** (go to the next tab position), **[ESC]** (return to the command loop) and **[CTRL]P** (toggle the printer) may be employed. The **[DEL]** key will produce a destructive backspace (but not beyond the beginning of the text line). Text is entered into an internal buffer within the run times and if this buffer should become full then you will be prevented from entering any more text - you must then use **[DEL]** or **[CTRL]X** to free space in the buffer.

Command: `I n,m`

Use of this command gains entry to the automatic insert mode: you are prompted with line numbers starting at `n` and incrementing in steps of `m`. You enter the required text after the displayed line number, using the various control codes if desired and terminating the text line with **[ENTER]**. To exit from this mode use the **[ESC]** key.

If you enter a line with a line number that already exists in the text then the existing line will be renumbered to a number one greater than previously and the line you typed in will be inserted with the existing line number after you have pressed **[ENTER]**. If the automatic incrementing of the line number produces a line number greater than 32767 then the Insert mode will exit automatically.

If, when typing in text, you get to the end of a screen line without having entered 80 characters (the buffer size) then the screen will be scrolled up and you may continue typing on the next line.

## 4.2.2 Text Listing.

You may direct a listing of your program to either the Amstrad screen (via the '**L**' command) or to the printer (via the '**Z**' command).

Command: **L** n,m

This lists the current text to the screen from line number n to line number m inclusive. The default value for n is always 1 and the default value for m is always 32767 i.e. default values are not taken from previously entered arguments. To list the entire textfile simply use '**L**' without any arguments. The listing will take place a page (24 lines) at a time; after displaying a page the list will pause (if not yet at line number m), hit **[ESC]** to return to the main editor loop or any other key to continue the listing.

Command: **Z** n,m

List the textfile to the attached printer. If no printer is connected to the computer or the printer is currently off-line then the message `NO PRINTER!` will be displayed and no action taken; otherwise the textfile, between line numbers n and m inclusive, will be printed.

If n and m are not specified then the whole textfile will be printed.

You may pause the listing, while printing, by hitting any key on the keyboard; subsequently hit **[ESC]** to return to the editor and abort the print or any other key to resume printing.

## 4.2.3 Text Editing.

Once some text has been created there will inevitably be a need to edit some lines. Various commands are provided to enable lines to be amended, deleted, moved and renumbered. Various commands exist to achieve this and these are given below. Some elementary form of screen editing is also supported and this works as follows:

Whenever you are in the command mode of the editor (i.e. with a '>' sign in the left margin of the current line) then you may split the cursor into a read cursor and a write cursor by holding the **[SHIFT]** key down together with one of the cursor keys. The write cursor will remain in the same position as the original cursor while you can move the read cursor about the screen (but not off the screen) using **[SHIFT]** and the cursor keys. Release **[SHIFT]** and the relevant cursor key when the read cursor is where you want it to be. You can now either type directly on the keyboard and the characters will appear at the write cursor or you can press the **[COPY]** key and in this case characters will be transferred from the read cursor position to the write cursor position and both cursor positions will be incremented.

To terminate this screen copy mode simply press **[ENTER]**, the read cursor will disappear and the line containing the write cursor will be scanned normally by the editor.

In addition to this screen-editing capability, various line-editing commands are supported viz:

Command: **D** <n,m>

All lines from n to m inclusive are deleted from the textfile. If m<n or less than two arguments are specified then no action will be taken; this is to help prevent careless mistakes. A single line may be deleted by making m=n ; this can also be accomplished by simply typing the line number followed by **[ENTER]**.

Command: **M** <n,m,d>

Moves the block of text between line numbers n and m inclusive to a position before the line with line number d and deletes the original block of text. The block that is moved will be renumbered starting with a line number that is 1 greater than the line number preceeding d.
Lines cannot be moved within themselves so that d must not lie within the block of lines n to m.

Command: **N** <n,m>

Use of the '**N**' command causes the textfile to be renumbered with a first line number of n and in line number steps of m. Both n and m must be present and if the renumbering would cause any line number to exceed 32767 then the original numbering is retained.

Command: **F** n,m,f,s

The text existing within the line range n < x < m is searched for an occurrence of the string f - the 'find' string. If such an occurrence is found then the relevant text line is displayed and the Edit mode is entered - see below. You may then use commands within the Edit mode to search for subsequent occurrences of the string f within the defined line range or to substitute the string s (the 'substitute' string) for the current occurrence of f and then search for the next occurrence of f; see below for more details.

Note that the line range and the two strings may have been set up previously by any other command so that it may only be necessary to enter '**F**' to initiate the search - see the example in Section 4.3 for clarification.

Command: **E** n

Edit the line with line number n. If n does not exist then no action is taken; otherwise the line is copied into a buffer and displayed on the screen (with the line number), the line number is displayed again underneath the line and the Edit mode is entered. All subsequent editing takes place within the buffer and not in the text itself; thus the original line can be recovered at any time.
In this mode a pointer is imagined moving through the line (starting at the first character) and various sub-commands are supported which allow you to edit the line. The sub-commands are:

' ' (space) - increment the text pointer by one i.e. point to the next character in the line. You cannot step beyond the end of the line.

**[DEL]** - decrement the text pointer by one to point at the previous character in the line. You cannot step backwards beyond the first character in the line.

**[TAB]** - step the text pointer forwards to the next tab position but not beyond the end of the line.

**[ENTER]** - end the edit of this line keeping all the changes made.

**Q** - quit the edit of this line i.e. leave the edit ignoring all the changes made and leaving the line as it was before the edit was initiated.

**R** - reload the edit buffer from the text i.e. forget all changes made on this line and restore the line as it was originally.

**L** - list the rest of the line being edited i.e. the remainder of the line beyond the current pointer position. You remain in the Edit mode with the pointer re-positioned at the start of the line.

**K** - kill (delete) the character at the current pointer position.

**Z** - delete all the characters from (and including) the current pointer position to the end of the line.

**F** - find the next occurrence of the '`find`' string previously defined within a command line (see the '**F**' command above). This sub-command will automatically exit the edit on the current line (keeping the changes) if it does not find another occurrence of the '`find`' string in the current line. If an occurrence of the '`find`' string is detected in a subsequent line (within the previously specified line range) then the Edit mode will be entered for the line in which the string is found. Note that the text pointer is always positioned at the start of the found string after a successful search.

**S** - substitute the previously defined '`substitute`' string for the currently found occurrence of the 'find' string and then perform the sub-command '**F**' i.e. search for the next occurrence of the 'find' string. This, together with the above '**F**' sub-command, is used to step through the textfile optionally replacing occurrences of the '`find`' string with the '`substitute`' string - see Section 4.3 for an example.

**I** - insert characters at the current pointer position. You will remain in this sub-mode until you press **[ENTER]** - this will return you to the main Edit mode with the pointer positioned after the last character that you inserted. Using **[DEL]** within this sub-mode will cause the character to the left of the pointer to be deleted from the buffer while the use of **[TAB]** will advance the pointer to the next tab position, inserting spaces. A '`*`' cursor will be displayed while in this mode.

**X** - this advances the pointer to the end of the line and automatically enters the insert sub-mode detailed above.

**C** - change sub-mode. This allows you to overwrite the character at the current pointer position and then advances the pointer by one. You remain in the change sub-mode until you press **[ENTER]** whence you are taken back to the Edit mode with the pointer positioned after the last character you changed. **[DEL]** within this sub-mode simply decrements the pointer by one i.e. moves it left while **[TAB]** has no effect. A '`+`' cursor will be displayed while in this mode.

## 4.2.4 Tape Commands.

Text may be saved to tape or loaded from tape using the commands '**P**' and '**G**' and verified using '**V**':

Command: **P** `n,m,s`

The line range defined by `n < x < m` is saved under the filename specified by the string `s`. Remember that these arguments may have been set by a previous command.

Before entering this command make sure that your tape recorder is switched on and in `RECORD` mode.

Command: **G**,,`s`

The storage device is searched for a textfile with a filename of s. If the required file cannot be found then an error message will be displayed. If the correct file is found then it will be loaded into memory. If an error is detected during the load then an error message will be displayed and the load aborted. If this happens you must redo the command.

While searching is in progress you may abort the load by holding **[ESC]** down; this will interrupt the load and return to the main editor loop.

Note that if any textfile is already present then the textfile that is loaded will be appended to the existing file and the whole file will be renumbered starting with line 1 in steps of 1.

Command: **V**,,`s`

Verify a textfile.

The storage device is searched for a filename as specified by the string `s`. When found the file is compared with the textfile currently held in memory by the editor. If the match is exact then `VERIFIED` is displayed, otherwise the message `FAILED` is shown.

Command: **S**`n`

Set the speed at which files are dumped to tape.

Normally, tape files are saved at the slower speed of 1000 baud; you may change this to the high speed of 2000 baud by specifying a non-zero number after the **S** command. To return to the slower speed simply use **S** without a following number.

E.g. **S** `1` for high-speed tape saving
**S** for normal tape saving

## 4.2.5 Compiling and Running from the Editor.

Command: **A**

Alter the compile/run defaults.

`Press A [ENTER]`

and you will be asked to specify:

`Symbol Table size?`

you can now enter a decimal number, followed by **[ENTER]**, to specify a new value for the size of the compiler's symbol table. The value of the symbol table size at the beginning of a session is normally 1858 and this should be sufficient for compiling most programs.

If you simply hit **[ENTER]** instead of specifying a number then the symbol table size is not changed from its previous value.

Once you have dealt with the symbol table size you will then be prompted with:

`Translate Stack?`

You may now specify (by entering a decimal number followed by [ENTER]) the address of the stack that is to be used by any Pascal object code program that is translated using the 'T' command (see below). By default this stack is set to the value specified in answer to the RAM top question which you answered when entering the Pascal.

You will find it useful to set the Translate stack if you should need to reserve memory at the top of the memory space for routines that you wish to interface to your translated program.

If you hit **[ENTER]** without specifying a number then the Translate stack is not changed from its previous value.

Command: **C n**

This causes the text starting at line number `n` to be compiled. If you do not specify a line number then the text will be compiled from the first existing line. For further details see Section 0.2.

Command: **R**

The previously compiled object code will be executed, but only if the source has not been expanded in the meantime - see Section 0.2 for more detail.

Command: **T n**

This is the 'T'ranslate command. The current source is compiled from line `n` (or from the start if `n` is omitted) and, if the compilation is successful, you will be prompted with '`Ok?`': if you answer 'Y' to this prompt then the object code produced by the compilation will be moved to the end of the runtimes (destroying the compiler) and then the runtimes and the object code will be dumped out to tape with a filename equal to that previously defined for the '`find`' string. The code is dumped out in binary file format so that it may be subsequently loaded using the CPC464's BASIC `LOAD` command.

Translated object code contains instructions that perform a `MC START PROGRAM` call when the code is executed, thus the object code takes over the machine, clearing out BASIC and any RSXs that have been introduced to the firmware. Thus, if you require any RSXs to be present when your object code is running, you must initialise these yourself within your code by using the Pascal `USER` procedure to call the firmware routine `KL LOG EXT (#BCD1)` to introduce the relevant RSX.

Note that the object code is located at and moved to the end of the runtimes so that, after a '**T**'ranslate you will need to reload the compiler - however this should present no problems since you are only likely to '**T**'ranslate a program when it is fully working.

If you decide not to continue with the dump to tape then simply type any character other than 'Y' to the '`Ok?`' prompt; control is returned to the editor which will still function perfectly since the object code was not moved.

## 4.2.6 Other Commands.

Command: **H**

This command displays a help screen of the various editing commands available to you. The commands are given in capital letters.

Command: **Q**,,d

This command allows you to change the delimiter which is taken as separating the arguments in the command line. On entry to the editor the comma '**,**' is taken as the delimiter; this may be changed by the use of the '**Q**' command to the first character of the specified string d. Remember that once you have defined a new delimiter it must be used (even within the '**Q**' command) until another one is specified.

Note that the separator may not be a space.

Command: **U**

Simply displays the last line number in your current textfile. This command is useful for finding the end of the textfile so that you can easily append to or list the end of the file.

Command: **W**

Flips the screen mode between 40 characters per line and 80 characters per line. The screen is initially set up to be in mode 1 i.e. 40 characters per line. Change this to 80 per line using **W [ENTER]** once and then you can change it back to 40 per line by using **W [ENTER]** a second time.

Command: **Y**

This command takes no arguments and it simply displays the current default values of the command delimiter, line range (`n-m`) and Find and Substitute strings. It should be remembered that certain editor commands (like '**D**' and '**L**') do not use the default line range but must have values specified on the command line.

Command: **X**

This displays the addresses of the start and end of text in hexadecimal. Useful for finding out the size of your textfile in bytes.

Command: **|**

The vertical bar ('**|**') command allows you to invoke background ROM commands from within the editor.

The bar must be followed by a valid command name for the external ROM and this may be optionally followed by any arguments needed by the command. Arguments must be separated by a comma and string arguments must be enclosed within single quotes ('**'**').

If the command or the arguments given are invalid then the error message '`Pardon`' will be displayed.

E.g. `|dir,'*.PAS'`

to obtain a directory listing of all the disc files with an extension of PAS.

```
|basic
```

to return to BASIC.

## 4.3 An Example of the use of the Editor.

Let us assume that you have typed in the following program (using I10,10):

```
10  PROGRAM BUBBLESORT
20  CONST
30  Size = 2000;
40  VAR
50  Numbers : ARRAY [1..Size] OF INTEGER;
60  I, Temp : INTEGER;
70  BEGIN
80  FOR I:=1 TO Size DO Number[I] :=RANDOM;
90  REPEAT
100 FOR I:=1 TO Size DO
110 Noswaps := TRUE;
120 IF Number[I] > Number[I+1] THEN
130 BEGIN
140 Temp := Number[I];
150 Number[I] := Number[I+1];
160 Number[I+1] := Temp;
170 Noswaps := FALSE
180 END
190 UNTIL Noswapss;
195 FORI := 1 TO Size DO WRITE(Number[I]:4)
200 END.
```

This program has a number of errors which are as follows:

Line 10 Missing semi-colon.
Line 30 Not really an error but say we want a size of 100.
Line 100 Size should be Size-1.
Line 110 This should be at line 95 instead.
Line 190 Noswapss should be Noswaps.

Also the variable Numbers has been declared but all references are to Number. Finally the `BOOLEAN` variable `Noswaps` has not been declared.

To put all this right we can proceed as follows:

| | |
|---|---|
| `F60,210,`Number,Numbers | and then use the sub-command '**S**' repeatedly. |
| `E10` | then the sequence **X ; [ENTER] [ENTER]** |
| `E30` | then _________ K C 1 **[ENTER] [ENTER]** |
| `F100,100,Size,Size-1` | followed by the sub-command '**S**'. |
| `M110,95` | |
| `E190` | then **X DELETE [ENTER] [ENTER]** |
| `65 Noswaps : BOOLEAN;` | |
| `N10,10` | to renumber in steps of 10. |

You are strongly recommended to work through the above example actually using the editor.

# APPENDIX 1
# ERRORS.

## A.1.1 Error numbers generated by the compiler.

1. Number too large.
2. Semi-colon expected.
3. Undeclared identifier.
4. Identifier expected.
5. Use '=' not ':=' in a constant declaration.
6. '=' expected.
7. This identifier cannot begin a statement.
8. ':=' expected.
9. ')' expected.
10. Wrong type.
11. '.' expected.
12. Factor expected.
13. Constant expected.
14. This identifier is not a constant.
15. 'THEN' expected.
16. 'DO' expected.
17. 'TO' or 'DOWNTO' expected.
18. '(' expected.
19. Cannot write this type of expression.
20. 'OF' expected.
21. ',' expected.
22. ':' expected.
23. 'PROGRAM' expected.
24. Variable expected since parameter is a variable parameter.
25. 'BEGIN' expected.
26. Variable expected incall to READ.
27. Cannot compare expressions of this type.
28. Should be either type INTEGER or type REAL.
29. Cannot read this type of variable.
30. This identifier is not a type.
31. Exponent expected inreal number.
32. Scalar expression (notnumeric) expected.
33. Null strings not allowed (useCHR(Ø)).

34. '[' expected.
35. ']' expected.
36. Array index type must be scalar.
37. '..' expected.
38. ']' or ',' expected in ARRAY declaration.
39. Lowerbound greater than upperbound.
4Ø. Set too large (more than 256 possible elements).
41. Function result must be type identifier.
42. ',' or ']' expected in set.
43. '..' or ',' or ']' expected in set.
44. Type of parameter must be a type identifier.
45. Null set cannot be the first factor in a non-assignment statement.
46. Scalar (including real) expected.
47. Scalar (not including real) expected.
48. Sets incompatible.
49. '<' and '>' cannot be used to compare sets.
5Ø. 'FORWARD', 'LABEL', 'CONST', 'VAR', 'TYPE' or 'BEGIN' expected.
51. Hexadecimal digit expected.
52. Cannot POKE sets.
53. Array too large (>64K).
54. 'END' or ';' expected in RECORD definition.
55. Field identifier expected.
56. Variable expected after 'WITH'.
57. Variable in WITH must be of RECORD type.
58. Field identifier has not had asociated WITH statement.
59. Unsigned integer expected after 'LABEL'.
6Ø. Unsigned integer expected after 'GOTO'.
61. This label is at the wrong level.
62. Undeclared label.
63. The parameter of SIZE should be a variable.
64. Can only use equality tests for pointers.
67. The only write parameter for integers with two ':'s is e:m:H.
68. Strings may not contain end of line characters.
69. The parameter of NEW, MARK or RELEASE should be a variable of pointer type.
7Ø. The parameter of ADDR should be a variable.
71. This parameter must be a procedure.
72. This parameter must be a parameterless procedure.
73. No more than 5 sections in an envelope.

## A.1.2 Runtime Error Messages.

When a runtime error is detected then one of the following messages will be displayed, followed by `' at PC=XXXX'` where `XXXX` is the memory location at which the error occurred. Often the source of the error will be obvious; if not, consult the compilation listing to see where in the program the error occurred, using `XXXX` to cross reference. Occasionally this does not give the correct result.

```
1.Halt
2.Overflow
3.Out of RAM
4./by zero                 also generated by DIV.
5.Index too low
6.Index too high
7.Maths Call Error
8.Number too large
9.Number expected
10.Line too long
11.Exponent expected
```

Runtime errors result in the program execution being halted.

# APPENDIX 2 Reserved words and predefined identifiers.

## A 2.1 Reserved Words.

| | | | | | | |
|---|---|---|---|---|---|---|
| AND | ARRAY | BEGIN | CASE | CONST | DIV | DO |
| DOWNTO | ELSE | END | FORWARD | FUNCTION | GOTO | IF |
| IN | LABEL | MOD | NIL | NOT | OF | OR |
| PACKED | PROCEDURE | PROGRAM | RECORD | REPEAT | SET | THEN |
| TO | TYPE | UNTIL | VAR | WHILE | WITH | |

## A 2.2 Special Symbols.

The following symbols are used by Hisoft Pascal 4 and have a reserved meaning:

```
        +          -          *          /
        =          <>         <          <=         >=         >
        (          )          [          ]
        {          }          (*         *)
        ^          :=         .          ,          ;          :
        '          ..
```

## A 2.3 Predefined Identifiers.

The following entities may be thought of a declared in a block surrounding the whole program and they are therefore available throughout the program unless re-defined by the programmer within an inner block.

For further information see Section 2.

| | |
|---|---|
| CONST | MAXINT=32767; |
| TYPE | BOOLEAN=(FALSE,TRUE);<br>CHAR {The expanded ASCII character set;<br>INTEGER=-MAXINT..MAXINT;<br>REAL {A subset of the real numbers. See Section 1.3. |
| VAR | ERRFLG, ERRCHK: BOOLEAN: RA, RB, RC, RD, RE, RF, RH, RL: CHAR;<br>RAF, RBC, RDE, RHL: INTEGER; |
| PROCEDURE | WRITE; WRITELN; READ; READLN; PAGE;<br>HALT; USER; POKE; INLINE;<br>OUT; NEW; MARK; RELEASE; TIN; TOUT;<br>AFTER; EVERY; SOUND; ONSQ;<br>EXTERNAL; ENV; ENT; |
| FUNCTION | ABS; SQR; ODD; RANDOM; ORD; SU[C;<br>PRED; INCH; EOLN;<br>PEEK; CHR; SQRT; ENTIER; ROUND;<br>TRUNC; FRAC; SIN; COS;<br>TAN; ARCTAN; EXP; LN; ADDR; SIZE;<br>INP; REMAIN; INITEVENT; REMAIN; |

# APPENDIX 3 Data representation and storage.

## A 3.1 Data Representation.

The following discussion details how data is represented internally by Hisoft Pascal.

The information on the amount of storage required in each case should be of use to most programmers (the `SIZE` function may be used see Section 2.3.6.7); other details may be needed by those attempting to merge Pascal and machine code programs.

### A 3.1.1 Integers.

Integers occupy 2 bytes of storage each, in 2's complement form. Examples:

```
   1    ≡    #ØØØ1
 256    ≡    #Ø1ØØ
-256    ≡    #FFØØ
```

The standard Z80 register used by the compiler to hold integers is HL.

### A 3.1.2 Characters, Booleans and other Scalars.

These occupy 1 byte of storage each, in pure, unsigned binary.

Characters: 8 bit, extended ASCII is used.

```
'E'    ≡    #45
'['    ≡    #5B
```

Booleans:

`ORD(TRUE)=1` so `TRUE` is represented by `1`.
`ORD(FALSE)=Ø` so `FALSE` is represented by `Ø`.

The standard Z80 register used by the compiler for the above is A.

## A 3.1.3 Reals.

The (mantissa, exponent) form is used similar to that used in standard scientific notation -only using binary instead of denary. Examples:

$2 \equiv 2*10^0$ or $1.0_2*2^1$

$1 \equiv 1*10^0$ or $1.0_2*2^0$

$-12.5 \equiv -1.25*10^1$ or $-25*2^{-1}$

$\equiv -11001_2*2^{-1}$

$-1.1001_2*2^3$ when normalised.

$0.1 \equiv 1.0*10^{-1}$ or $\frac{1}{10} \equiv \frac{1}{1010_2} \equiv \frac{0.1_2}{101_2}$

so now we need to do some binary long division..

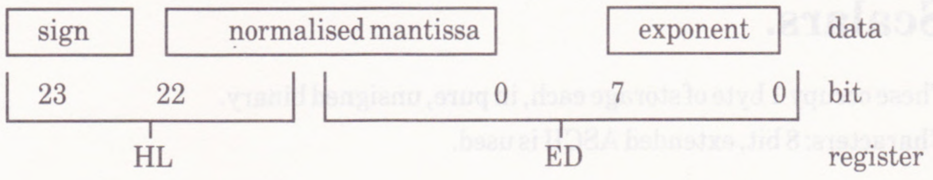
```
          0.0001100
     ____________________
101 | 0.1000000000000000
          101
          110
          101
          ____
            1000
             101
             ___
```

at this point we see that the fraction recurs

$$= \frac{0.1_2}{101_2} = 0.000\dot{1}10\dot{0}_2$$

$$\underline{1.\dot{1}00\,11\dot{0}0*2^{-4}}$$ answer.

So how do we use the above results to represent these numbers in the computer? Well, firstly we reserve 4 bytes of storage for each real in the following format:



| | |
|---|---|
| sign: | the sign of the mantissa; 1 = negative, 0 = positive. |
| normalised mantissa: | the mantissa normalised to the form 1.xxxxxx with the top bit (bit 22) always 1 except when representing zero (HL=0, DE=0). |
| exponent: | the exponent in binary 2's complement form. |

Thus:

```
2 ≡       Ø 1000000 ØØØØØØØØ ØØØØØØØØ ØØØØØØØ1   (#4Ø,#ØØ,#ØØ,#Ø1)
1 ≡       Ø 1000000 ØØØØØØØØ ØØØØØØØØ ØØØØØØØØ   (#4Ø,#ØØ,#ØØ,#ØØ)
-12.5 ≡   1 11ØØ1ØØ ØØØØØØØØ ØØØØØØØØ ØØØØØØ11   (#E4,#ØØ,#ØØ,#Ø3)
Ø.1 ≡     Ø 11ØØ11Ø Ø11ØØ11Ø Ø11ØØ11Ø 111111ØØ   (#66,#66,#66,#FC)
```

So, remembering that HL and DE are used to hold real numbers, then we would have to load the registers in the following way to represent each of the above numbers:

```
2          ≡      LD  HL,#4ØØØ
                  LD  DE,#Ø1ØØ
1          ≡      LD  HL,#4ØØØ
                  LD  DE,#ØØØØ
-12.5      ≡      LD  HL,#E4ØØ
                  LD  DE,#Ø3ØØ
Ø.1        ≡      LD  HL,#6666
                  LD  DE,#FC66
```

N.B. Reals are stored in memory in the order ED LH.

## A 3.1.4 Records and Arrays.

Records use the same amount of storage as the total of their components.

Arrays: if `n` = number of elements in the array and
`s` = size of each element then

the number of bytes occupied by the array is `n*s`.

e.g. an `ARRAY[1..1Ø] OF INTEGER` requires `1Ø*2=2Ø` bytes an `ARRAY [2..12,1..1Ø] OF CHAR` has `11*1Ø=11Ø` elements and so requires 110 bytes.

## A 3.1.5 Sets.

Sets are stored as bit strings and so if the base type has `n` elements then the number of bytes used is: `(n-1) DIV 8 + 1`. Examples:
a `SET OF CHAR` requires `(256-1) DIV 8 + 1 = 32` bytes a `SET OF` (blue, green, yellow) requires `(3-1) DIV 8 + 1 = 1` byte

## A 3.1.6 Pointers.

Pointers occupy 2 bytes which contain the address (in Intel format i.e. low byte first) of the variable to which they point.

## A 3.2 Variable Storage at Runtime.

There are 3 cases where the user needs information on how variables are stored at runtime:

| | |
|---|---|
| a. Global variables | - declared in the main program block. |
| b. Local variables | - declared in an inner block. |
| c. Parameters and returned values. | - passed to and from procedures and functions. |

These individual cases are discussed below and an example of how to use this information may be found in Appendix 4.

**Global variables**

Global variables are allocated from the top of the runtime stack downwards e.g. if the runtime stack is at `#BØØØ` and the main program variables are:

```
VAR     i  :INTEGER;
        ch :CHAR;
        x  :REAL;
```

then:

`i` (which occupies 2 bytes - see the previous section) will be stored at locations `#BØØØ-2` and `#BØØØ-1` i.e. at `#AFFE` and `#AFFF`.

`ch` (1 byte) will be stored at location `#AFFE-1` i.e. at `#AFFD`.

`x` (4 bytes) will be placed at `#AFF9`, `#AFFA`, `#AFFB` and `#AFFC`.

**Local variables**

Local variables cannot be accessed via the stack very easily so, instead, the IX register is set up at the beginning of each inner block so that (IX-4) points to the start of the block's local variables e.g.

```
PROCEDURE     test;
VAR           i,j :INTEGER;
```

then:

i (integer - so 2 bytes) will be placed at IX-4-2 and IX-4-1 i.e. IX-6 and IX-5. j will be placed at IX-8 and IX-7.

## Parameters and returned values

Value parameters are treated like local variables and, like these variables, the earlier a parameter is declared the higher address it has in memory. However, unlike variables, the lowest (not the highest) address is fixed and this is fixed at (IX+2) e.g.

```
PROCEDURE test(i : REAL; j : INTEGER);
```

then:

j (allocated first) is at IX+2 and IX+3.
i is at IX+4, IX+5, IX+6, and IX+7.

Variable parameters are treated just like value parameters except that they are always allocated 2 bytes and these 2 bytes contain the address of the variable e.g.

```
PROCEDURE test(i : INTEGER; VAR x : REAL);
```

then:

the reference to x is placed at IX+2 and IX+3; these locations contain the address where x is stored. The value of i is at IX+4 and IX+5.

**Returned values** of functions are placed above the first parameter in memory e.g.

```
FUNCTION test(i : INTEGER) : REAL;
```

then i is at IX+2 and IX+3 and space is reserved for the returned value at IX+4, IX+5, IX+6 and IX+7.

# APPENDIX 4
# Some example HiSoft Pascal programs.

The following programs should be studied carefully if you are in any doubt as to how to program in Hisoft Pascal.

```
PROGRAM FACTOR

10   (*Program to show the use of recursion*)
20
30   PROGRAM FACTOR;
40
50   (*This  program  calculates  the  factorial  of  a
      number input from the
60   keyboard  1)  using  recursion  and  2)  using  an
      iterative method.*)
70
80   TYPE
90   POSINT = 0..MAXINT;
100
110  VAR
120  METHOD : CHAR;
130  NUMBER : POSINT;
140
150  (*Recursive algorithm.*)
160
170  FUNCTION RFAC(N : POSINT) : INTEGER;
180
190  VAR F : POSINT;
200
210  BEGIN
220  IF  N>1  THEN  F:=  N  *  RFAC(N-1)  (*RFAC  invoked  N
      times*)
230  ELSE F:=1;
240  RFAC := F
250  END;
260
270  (*Iterative solution*)
280
290 FUNCTION IFAC(N : POSINT) : INTEGER;
```

```
300
310 VARI,F:POSINT;
320 BEGIN
330  F:=1;
340  FORI:=2TONDOF:=F*I;(*SimpleLoop*)
350  IFAC:=F
360 END;
370
380 BEGIN
390 REPEAT
400 WRITE('Give method (I or R) and number');
410 READLN;
420 READ(METHOD,NUMBER);
430 IF METHOD = 'R'
440 THEN WRITELN(NUMBER,'! = ',RFAC(NUMBER))
450 ELSE WRITELN(NUMBER,'! = ',IFAC(NUMBER))
460 UNTIL NUMBER=Ø
470 END.
```

PROGRAMREV

```
10  {Program to list lines of a file in reverse order.
20  Shows use of pointers, records, MARK and RELEASE.}
30
40  PROGRAM ReverseLine;
50
60  TYPEelem=RECORD               {Create linked-list structure}
70  next: ↑elem;
80  ch: CHAR
90  END;
100 link=↑elem;
110
120 VARprev,cur,heap: link; {all pointers to'elem'}
130
140 BEGIN
150 REPEAT                                 {do this many times}
160 MARK(heap);                   {assign top of heap to 'heap'}
170 prev:=NIL;                      {points to no varaible yet.}
180 WHILE NOT EOLN DO
190 BEGIN
200 NEW(cur);                      {create a new dynamic record}
210 READ(cur↑ch);                   {and assign its field to one
220                                          character from file.}
230 cur↑next:=prev;      {this field's pointer adresses}
240 prev:=cur{previous record.}
250 END;
260
270 {Write out the line backwards by scanning
280 the records set up backwards.}
290
300 cur:=prev;
310 WHILEcur<>NILDO                                {NIL is first}
320 BEGIN
330 WRITE(cur↑.ch);        {WRITE this field i.e. character}
340 cur:=cur↑.next                  {Address previous field.}
350 END;
360 WRITELN;
370 RELEASE(heap);        {Release dynamic variable space,}
380 READLN {Waitforanotherline}
390 UNTIL FALSE {Use [ESC] to exit}
400 END.
```

## PROGRAM TINTOUT

Program to illustrate the use of `TIN` and `TOUT`. The program constructs a very simple telephone directory on tape and then reads it back. You should write any searching required.

```
PROGRAM TINTOUT;

CONST
  Size=10;
TYPE
  Entry=RECORD
     Name:ARRAY[1..10]OF CHAR;
     Number:ARRAY[1..10]OF CHAR
   END;
VAR
 Directory:ARRAY[1..Size]OF Entry;
 I:INTEGER;
BEGIN
(Set up the directory..
FOR I:= 1 TO Size DO
BEGIN
 WITH Directory[I] DO
 BEGIN
    WRITE('Name please');
    READLN;
    READ(Name);
    WRITELN;
    WRITE('Number please');
    READLN;
    READ(Number);
    WRITELN
 END
END;
{To dump the directory to tape use..}
TOUT('Directory',ADDR(Directory),SIZE(Directory))
{Now to read the array back do the following..}
TIN('Directory',ADDR(Directory))
{And now you can process the directory as
you wish....}
END.
```

PROGRAM DIRTY.

```
10   {Program to show how to 'get your hands dirty'!
20   i.e. how to modify Pascal variables using machine code.
30   Demonstrates PEEK, POKE, ADDR and INLINE}
40
50   PROGRAM divmult2;
60
70   VAR r:REAL;
80
90   FUNCTION divby2(x:REAL):REAL;                    {Function to
                                                       divide by 2..
100                                                    ..quickly}
110 VAR i:INTEGER;
120 BEGIN
130 i:=ADDR(x)+1;               {Point to the exponent of x}
140 POKE(i,PRED(PEEK(i,CHAR)));                  {Decrement the
                                                  exponent of x}
150 {see Appendix 3.1.3}
160 divby2:=x
170 END;
180
190 FUNCTION multby2(x:REAL):REAL;                   {Function to
                                                  multiply by 2..
200                                                    ..quickl}y
210 BEGIN
220 INLINE(#DD,#34,3);      {INC (IX+3) - the exponent of x
230                                   - see Appendix 3.2}
240 multby2:=x
250 END;
260
270 BEGIN
280 REPEAT
290 WRITE('Enter the number r ');
300 READ(r);                         {No need for READLN - see
310                                            Section 2.3.1.4}
320
330 WRITELN('r divided by two is',divby2(r):7:2);
340 WRITELN('r multiplied by two is',multby2(r):7:2)
350 UNTIL r=0
360 END.
```

# APPENDIX 5 HiSoft PASCAL Turtle Graphics

The Turtle Graphics package is contained on the reverse side of your master cassette tape under the name TURTLE.

The package is written in Pascal and may be loaded from within the Hisoft Pascal editor by using the command 'G,,TURTLE'. This will load the turtle graphics program segment and append it to any existing program; note that, in order for it to function correctly, the Turtle Graphics must be preceded by a normal PROGRAM heading and a VAR declaration - TYPE, CONST and LABEL declarations are optional and there must be no Procedures or Functions declared previous to the inclusion of the Turtle Graphics package.

The TURTLE package as supplied contains a demonstration program and to run this you should simply:

```
g,,TURTLE[ENTER]
C[ENTER]
```

and answer y to the Run? question at the end of the compilation.

To extract the core Turtle Graphics routines, which are documented below, you should:

```
d10,40[ENTER]
d1150,2320[ENTER]
p1,1140,turtle[ENTER]
```

although you may, of course, want to keep some of the other procedures and functions that are part of the demonstration program; these are placed between line numbers 1150 through to 2320.

As in the majority of Turtle Graphics implementations, Hisoft Pascal's TURTLE creates an imaginary creature on the screen which the user can move around via some very simple commands. This 'turtle' can be made to leave a trail (in varying colours) or can be made invisible. The turtle's heading and position are held in global variables which are updated when the creature is moved or turned; obviously these variables can be inspected or changed at any time.

The facilities available are as follows:

## Global Variables

### heading

this is used to hold the angular value of the direction in which the turtle is currently facing. It takes any `REAL` value, in degrees, and may be initialised to `Ø` with the procedure `TURTLE` (see below). The value `Ø` corresponds to an `EAST`erly direction so that after a call to the procedure `TURTLE` the turtle is facing left to right. As the heading increases from zero then the turtle turns in an anti-clockwise direction.

### Xcor, Ycor

these are the current `(x,y)` `REAL` co-ordinates of the turtle on the screen. The CPC464 graphics screen has a size of `640*200` pixels and the turtle may be positioned on any point within this area assuming you are working in the highest resolution mode; when using lower resolution modes you may still specify `640*400` but the resolution will not be one dot.

Initially `Xcor` and `Ycor` are undefined, use of the procedure TURTLE initialises them to `300` and `200` respectively, thus placing the turtle in the middle of his 'pool'.

### penstatus

an `BOOLEAN` variable holding the current status of the '`pen`' (i.e. the trail left by the turtle). `TRUE` means the pen is down, `FALSE` means the pen is up.

## Procedures

The procedures available are:

### INK (I,C1,C2:INTEGER)

sets the ink `I` to have the colour values specified by `C1` and `C2`. If `C1 = C2` then the ink will be a steady colour, otherwise the ink will be flashing.

E.g.

```
INK(1,12,12);        sets ink 1 to steady yellow
INK(Ø,16,21);        sets ink Ø to flashing pink and lime!
```

### PAPER (I:INTEGER)

sets the background (paper) colour of the screen to the colour(s) associated with the `ink I` which is an integer.

**PEN ( I:INTEGER);**

sets the turtle's pen colour to the colour(s) associated with the ink I.

**PENDOWN ( I:INTEGER )**

sets the turtle state so that it will leave a trail in the ink colour associated with the parameter I.
This procedure assigns `TRUE` to penstatus.

**PENUP**

subsequent to a call to this procedure the turtle will not leave a trail. Useful for moving from one graphic section to another.
`PENUP` assigns the value `FALSE` to penstatus.

**SETHD ( A:REAL )**

takes a `REAL` parameter which is assigned to the global variable heading thus setting the direction in which the turtle is pointing. Remember that a heading of `Ø` corresponds to `EAST`, `9Ø` to `NORTH`, `18Ø` to `WEST` and `27Ø` to `SOUTH`.

**SETXY ( X,Y:REAL )**

sets the absolute position of the turtle within the graphics area to the value `(X,Y)`. No check is made within this procedure to ascertain if `(X,Y)` is out of bounds; the firmware checks for this.

**FWD ( L:REAL )**

moves the turtle forward `L` units in the direction of its current heading. A unit corresponds to a graphics pixel, rounded up or down where necessary.

**BACK ( L:REAL )**

moves the turtle `L` units in the directly opposite direction to that in which it is currently heading (i.e. `-18Ø`) - the heading is left unchanged.

**TURN ( A:REAL )**

changes the turtle's heading by `A` degrees without moving it. The heading is increased in the anti-clockwise direction.

**MODE (M:INTEGER)**

this sets the screen to mode `M`, where `M` is an integer with a value between `Ø` and 2 corresponding to the screen mode established in BASIC.

**RIGHT ( A:REAL )**

an alternative to `TURN - RIGHT` changes the turtle's heading in the clockwise direction by `A` degrees.

**LEFT ( A:REAL )**

this is identical to TURN and is provided simply for convenience and compatibility with RIGHT.

**ARCR ( R:REAL, A:INTEGER )**

the turtle moves through an arc of a circle whose size is set by R. The length of the arc is determined by A, the angle turned through (subtended at the centre of the circle) in a clockwise direction. Typically R may be set to Ø.5.

**TURTLE**

this procedure simply sets the initial state of the turtle; it is placed in the middle of the screen, facing EAST (heading of Ø), on a bright yellow backgound paper and leaving a blue trail. Remember that the state of the turtle is not initially defined so that this procedure is often used at the beginning of a program.

This concludes the list of facilities available with Hisoft Pascal TURTLE; although simple in implementation and use you will find that Turtle Graphics are capable of prducing very complex designs at high speed. To give you a taste of this we present some example programs below. Remember that you must have Hisoft Pascal loaded before entering the programs.

## Example Programs

In all the example programs given below we assume that you have already loaded Hisoft Pascal and used 'G,,TURTLE' to load the Turtle Graphics package which starts at line 1Ø and finishes at line 114Ø and that you have deleted the demonstration-dependent parts of the package as detailed above. Now proceed with the examples:

**1. CIRCLES**

```
   1 PROGRAM CIRCLES;
   2 VAR I:INTEGER;

1150 BEGIN
1160 TURTLE;
1170 FOR I:=1 TO 9 DO
1180 BEGIN
1190 ARCR(0.5,360);
1200 RIGHT(40)
1210 END
1220 END.
```

## 2. SPIRALS

```
  1 PROGRAM SPIRALS;
  2 VAR

1150 PROCEDURE SPIRALS ( L,A:REAL );
1160 BEGIN
1170 FWD(L);
1180 RIGHT(A);
1190 SPIRALS(L+1,A)
1200 END;
1210 BEGIN
1220 TURTLE;
1230 SPIRALS(9,95)  (*or (9,90) or (9,121)...*)
1240 END.
```

## 3. FLOWER

```
  1 PROGRAM FLOWER;
  2 VAR

1150 PROCEDURE PETAL (S:REAL);
1160 BEGIN
1170 ARCR(S,60);
1180 LEFT(120);
1190 ARCR(S,60);
1200 LEFT(120)
1210 END;
1220 PROCEDURE FLOWER (S:REAL);
1230 VAR I:INTEGER;
1240 BEGIN
1250 FOR I =1 TO 6 DO
1260 BEGIN
1270 PETAL(S);
1280 RIGHT(60)
1290 END
1300 END;
1310 BEGIN TURTLE;
1320 SETXY(127,60);
1330 LEFT(90); FWD(10);
1340 RIGHT(60); PETAL(0.2);
1350 LEFT(60); PETAL(0.2);
1360 SETHD(90); FWD(40);
1370 FLOWER(0.4)
1380 END.
```

For further, extended study of Turtle Graphics we highly recommend the excellent (if expensive) book 'Turtle Geometry' by Harold Abelson and Andrea di Sessa, published by MIT Press, ISBN 0-262-01063-1.

# APPENDIX 6 Useful Routines for CPC464 Firmware

Given below is a listing of a set of Pascal procedures and functions that enable you to call various firmware routines from within Hisoft Pascal. You should select and use only the routines that you require for any one application. The routines are designed to be self-documenting.

## Library procedures for firmware calls

```
 10 (* getjoy is like BASIC's JOY FUNCTION; its parameter
 20     should be the 0 OR 1 ; it returns a "bit significant" value
 30     as IN BASIC *)
 40
 50 FUNCTION getjoy(stick:integer):integer;
 60 BEGIN
 70  user(#bb24);
 80  IF stick=1 THEN ra:=rl;
 90  getjoy:=ord(ra)
100 END;
110
120 (* txtinitialise re-initialises the Text VDU. That is
130    The text paper is SET TO ink 0.
140    The  text pen is SET TO ink 1.
150    The text window is SET TO the entire screen
160    The text cursor is enabled but turned off.
170    The character writing mode is SET TO opaque.
180    The VDU is enabled.
190    The graphic character write mode is turned off.
200    The cursor is moved TO the top left corner OF the window *)
210
220 PROCEDURE txtinitialise;
230 BEGIN
240  user(#bb4e)
250 END;
260
270 (* txtout outputs a character OR control code TO the
280    Text VDU without the Pascal processing control codes.
290    This should be used when, FOR example setting paper colours
300    when using control codes. *)
310
320 PROCEDURE txtout(c:char);
330 BEGIN
340  ra:=c;
350  user(#bb5a)
360 END;
370
380 (* txtrdchar reads a character from the screen at the current
390    cursor position. *)
400
410 FUNCTION txtrdchar:char;
420 BEGIN
430  user(#bb60);
440  txtrdchar:=ra
450 END;
460
```

```
(* winenable sets the size OF the current text window *)

PROCEDURE winenable(col1,col2,row1,row2:integer);
BEGIN
 rh:=chr(col1); rd:=chr(col2);
 rl:=chr(row1); re:=chr(row2);
 user(#bb66)
END;

 (* getwindow returns the size OF the current window IN its
    variable parameters *)

PROCEDURE getwindow(VAR col1,col2,row1,row2:integer);
BEGIN
 user(#bb69);
 col1:=ord(rh); col2:=ord(rd);
 row1:=ord(rl); row2:=ord(re)
END;

(* clearwindow clears the current text window *)

PROCEDURE clearwindow;
BEGIN
 user(#bb6c)
END;

(* setcolumn sets the cursor's horizontal position *)

PROCEDURE setcolumn(c:integer);
BEGIN
 ra:=chr(c);
 user(#bb6f)
END;

(* setrow sets the cursor's vertical position *)

PROCEDURE setrow(r:integer);
BEGIN
 ra:=chr(r);
 user(#bb72)
END;

(* setcursor sets the cursor's position TO the given
  column AND row *)

PROCEDURE setcursor(c,r:integer);
BEGIN
 rh:=chr(c); rl:=chr(r);
 user(#bb75)
END;
```

```
(* getcursor returns the current cursor position AND the
    current roll count. This roll count has no absolute
    meaning but is incremented IF the window is rolled down
    AND decremented IF it is rolled up. *)

PROCEDURE getcursor(VAR col,row,roll:integer);
BEGIN
 user(#bb78);
 col:=ord(rh); row:=ord(rl); roll:=ord(ra)
END;

(* curenable enables the cursor. The cursor is
    displayed IF it is both enabled AND on. *)

PROCEDURE curenable;
BEGIN
 user(#bb7b);
END;

(* curdisable disables the cursor *)

PROCEDURE curdisable;
BEGIN
 user(#bb7e)
END;

(* curon turns the cursor on AND displays it IF enabled *)

PROCEDURE curon;
BEGIN
 user(#bb81)
END;

(* curoff turns the cursor off *)

PROCEDURE curoff;
BEGIN
 user(#bb84)
END;


(* txtsetpen sets the pen FOR writing characters *)

PROCEDURE txtsetpen(ink:integer);
BEGIN
 ra:=chr(ink);
 user(#bb90)
END;

```

```
(* txtgetpen returns the current text pen ink *)

FUNCTION txtgetpen:integer;
BEGIN
 user(#bb93);
 txtgetpen:=ord(ra)
END;

(* txtsetpaper sets the ink FOR writing text background *)

PROCEDURE txtsetpaper(ink:integer);
BEGIN
 ra:=chr(ink);
 user(#bb96)
END;

(* gettxtpaper returns the ink FOR writing background *)

FUNCTION gettxtpaper:integer;
BEGIN
 user(#bb99);
 gettxtpaper:=ord(ra)
END;

(* txtinverse exchanges the current pen AND text inks *)

PROCEDURE txtinverse;
BEGIN
 user(#bb9c)
END;

(* txtsetback causes transparent mode TO be used IF its
   parameter is true. i.e. does NOT write the background.
   IF its parameter is false THEN the opaque mode which
   writes the background is used. *)

PROCEDURE txtsetback(b:boolean);
BEGIN
 ra:=chr(ord(b));
 user(#bb9f)
END;

(* txtgetback returns true IF transparent mode is being
   used. It returns false IF opaque mode is being used. *)

FUNCTION txtgetback:boolean;
BEGIN
 user(#bba2);
 txtgetback:= ra = chr(1)
END;

```

```
1980 (* txtgetmatrix returns the address OF the character matrix
1990    corresponding TO the character given by its parameter. *)
2000
2010 FUNCTION txtgetmatrix(c:char):integer;
2020 BEGIN
2030  ra:=c;
2040  user(#bba5);
2050  txtgetmatrix:=rhl
2060 END;
2070
2080 (* txtsetmatrix copies the character matrix at its adr
2090 parameter TO be used as the matrix FOR its char parameter *)
2100
2110 PROCEDURE txtsetmatrix(c:char;adr:integer);
2120 BEGIN
2130  ra:=c;
2140  rhl:=adr;
2150  user(#bba8)
2160 END;
2170
2180 (* setmtable sets the user defined character matrix address
2190    TO its addr parameter. Its c parameter is used as the
2200    number coresponding TO the lowest character used IN the
2210    table. IF this  parameter is NOT IN the range 0 TO 255
2220    THEN the user defined character table is considered
2230    TO be empty. Normally arrays are used TO store such
2240    matrices AND the addr FUNCTION used TO pass the address
2250    OF the table. *)
2260
2270 PROCEDURE setmtable(c,adr:integer);
2280 BEGIN
2290  rde:=c; rhl:=adr;
2300  user(#bbab)
2310 END;
2320
2330 (* txtstrselect selects a current stream number which
2340    should be IN the range 0..7. Many attributes OF the
2350    text VDU may be SET independently on different streams. *)
2360
2370 PROCEDURE txtstrselect(s:integer);
2380 BEGIN
2390  ra:=chr(s);
2400  user(#bbb4)
2410 END;
2420
```

```
(* txtswapstreams swaps the stream descriptors OF two streams *)

PROCEDURE txtswapstreams(s1,s2:integer);
BEGIN
 rb:=chr(s1); rc:=chr(s2);
 user(#bbb7)
END;

(*graintialise initialises the graphics VDU that is
 Sets the graphic paper TO ink Ø
 Sets the graphic pen TO ink 1
 Sets the user origin TO the bottom left corner OF the screen
 Moves the current position TO the user origin
 SET the graphics window to cover the whole screen.
 The graphics window is NOT cleared. *)

PROCEDURE grainitialise;
BEGIN
 user(#bbba)
END;

(* gramoveabsolute moves the current position TO the absolute
   position given by its parameters. *)

PROCEDURE gramoveabsolute(x,y:integer);
BEGIN
 rde:=x; rhl:=y;
 user(#bbcØ)
END;

(* gramoverelative moves the current position relative TO the
   current position *)

PROCEDURE gramoverelative(x,y:integer);
BEGIN
 rde:=x; rhl:=y;
 user(#bbc3);
END;

(* graaskcursor returns the current graphics position IN its
   variable parameters. *)

PROCEDURE graaskcursor(VAR x,y:integer);
BEGIN
 user(#bbc6);
 x:=rde; y:=rhl
END;
```

```
(* grasetorigin sets the location OF the user origin AND
   moves the current position there *)

PROCEDURE grasetorigin(x,y:integer);
BEGIN
 rde:=x; rhl:=y;
 user(#bbc9)
END;

(* gragetorigin returns the position OF the user origin. *)

PROCEDURE gragetorigin(VAR x,y:integer);
BEGIN
 user(#bbcc);
 x:=rde; y:=rhl
END;

(* grawinwidth sets the right AND left edges OF the
   graphics window. *)

PROCEDURE grawinwidth(x1,x2:integer);
BEGIN
 rde:=x1; rhl:=x2;
 user(#bbcf)
END;

(* grawinheight sets the top AND bottom edges OF the
   graphics window *)


PROCEDURE grawinheight(y1,y2:integer);
BEGIN
 rde:=y1; rhl:=y2;
 user(#bbd2)
END;

(* gragetwwidth returns the left AND right edges OF the
   graphics window *)

PROCEDURE gragetwwidth(VAR x1,x2:integer);
BEGIN
 user(#bbd5);
 x1:=rde; x2:=rhl
END;

```

```
(* gragetwheight returns the top AND bottom OF the graphics
   window *)

PROCEDURE gragetwheight(VAR y1,y2:integer);
BEGIN
 user(#bbd8);
 y1:=rde; y2:=rhl
END;

(* graclearwindow clears the graphics window *)

PROCEDURE graclearwindow;
BEGIN
 user(#bbdb)
END;

(* grasetpen sets the gaphics pen ink which is used FOR
   plotting points AND lines. *)

PROCEDURE grasetpen(ink:integer);
BEGIN
 ra:=chr(ink);
 user(#bbde)
END;

(* gragetpen returns the current graphic plotting ink *)

FUNCTION gragetpen:integer;
BEGIN
 user(#bbe1);
 gragetpen:=ord(ra)
END;

(* grasetpaper sets the graphics background ink which is
   used FOR clearing the window. *)

PROCEDURE grasetpaper(ink:integer);
BEGIN
 ra:=chr(ink);
 user(#bbe4)
END;

 (* gragetpaper returns the current graphics background ink. *)

FUNCTION gragetpaper:integer;
BEGIN
 user(#bbe7);
 gragetpaper:=ord(ra)
END;

```

```
3860 (* graplotabsolute plots a point IN an absolute position IN
3870    the current graphics pen ink AND the current graphics
3880    write mode. *)
3890
3900 PROCEDURE graplotabsolute(x,y:integer);
3910 BEGIN
3920  rde:=x;rhl:=y;
3930  user(#bbea)
3940 END;
3950
3960 (* graplotrelative plots a point relative TO the current
3970    position IN the current graphics pen ink AND the current
3980    graphics write mode. *)
3990
4000 PROCEDURE graplotrelative(x,y:integer);
4010 BEGIN
4020  rde:=x;rhl:=y;
4030  user(#bbed)
4040 END;
4050
4060
4070 (* gratestabsolute moves the current graphics position AND
4080    returns the value OF the ink found there. *)
4090
4100 FUNCTION gratestabsolute(x,y:integer):integer;
4110 BEGIN
4120  rde:=x; rhl:=y;
4130  user(#bbf0);
4140  gratestabsolute:=ord(ra)
4150 END;
4160
4170 (* gratestrelative moves the graphics pointer relative TO
4180    the currrent position AND returns the value OF the ink
4190    at the new position. *)
4200
4210 FUNCTION gratestrelative(x,y:integer):integer;
4220 BEGIN
4230  rde:=x; rhl:=y;
4240  user(#bbf3);
4250  gratestrelative:=ord(ra)
4260 END;
4270
```

```
(* gralineabsolute moves the current position TO the endpoint
   supplied AND draws a line IN the current graphics ink
   using the current graphics mode. *)

PROCEDURE gralineabsolute(x,y:integer);
BEGIN
 rde:=x; rhl:=y;
 user(#bbf6)
END;

(* gralinerelative moves the current position TO the
   endpoint supplied AND draws a line IN the current graphics
   pen ink using the current graphics mode. *)

PROCEDURE gralinerelative(x,y:integer);
BEGIN
 rde:=x; rhl:=y;
 user(#bbf9)
END;

(* grawrchar writes a character on the screen at the current
   graphics position AND moves the graphics position right
   ready TO write a another character. *)

PROCEDURE grawrchar(c:char);
BEGIN
 ra:=c;
 user(#bbfc)
END;

(* scrinitialise re-initialises the Screen Pack, the mode,
   AND the inks. *)

PROCEDURE scrinitialise;
BEGIN
 user(#bbff)
END;

(* scrsetoffset sets the offset OF the first character OF the
   screen. by changing this offset the screen can be rolled
   by the hardware. *)

PROCEDURE scrsetoffset(ink:integer);
BEGIN
 rhl:=ink;
 user(#bc05)
END;

```

```
4760 (* scrgetlocation returns the current offset OF the first
4770    character on the screen. *)
4780
4790 FUNCTION scrgetlocation:integer;
4800 BEGIN
4810  user(#bc0b);
4820  scrgetlocation:=rhl
4830 END;
4840
4850 (* scrsetmode sets the screen IN a new mode; clears the
4860    screen AND sets up the windows TO be the whole screen *)
4870
4880 PROCEDURE scrsetmode(m:integer);
4890 BEGIN
4900  ra:=chr(m);
4910  user(#bc0e)
4920 END;
4930
4940 (* scrgetmode returns the current screen mode. *)
4950
4960 FUNCTION scrgetmode:integer;
4970 BEGIN
4980  user(#bc11);
4990  scrgetmode:=ord(ra)
5000 END;
5010
5020 (* scrclear clears the screen TO ink 0. *)
5030
5040 PROCEDURE scrclear;
5050 BEGIN
5060  user(#bc14)
5070 END;
5080
5090 (* scrcharlimits returns the last character column AND
5100    row on the screen IN the current mode. *)
5110
5120 PROCEDURE scrcharlimits(VAR col,row:integer);
5130 BEGIN
5140  user(#bc17);
5150  col:=ord(rb); row:=ord(rc)
5160 END;
5170
```

```
5180 (* scrsetink sets the colours TO display an ink. IF the
5190    two colours are the same THEN the ink will remain
5200    a steady colour. IF the colours are different THEN
5210    the ink will alternate between these two colours. *)
5220
5230 PROCEDURE scrsetink(ink,col1,col2:integer);
5240 BEGIN
5250  ra:=chr(ink); rb:=chr(col1); rc:=chr(col2);
5260  user(#bc32)
5270 END;
5280
5290 (* scrgetink returns the two coolours that are used TO
5300    display an ink on the screen. *)
5310
5320 PROCEDURE scrgetink(VAR col1,col2:integer);
5330 BEGIN
5340  user(#bc35);
5350  col1:=ord(rb); col2:=ord(rc)
5360 END;
5370
5380 (* scrsetborder sets the two colours TO display the border.
5390   IF the two colours are the same THEN the border will be
5400    displayed IN a steady colour. *)
5410
5420 PROCEDURE scrsetborder(col1,col2:integer);
5430 BEGIN
5440  rb:=chr(col1); rc:=chr(col2);
5450  user(#bc38)
5460 END;
5470
5480 (* scrgetborder returns the two colours used TO display the
5490    border. *)
5500
5510 PROCEDURE scrgetborder(VAR col1,col2:integer);
5520 BEGIN
5530  user(#bc3b);
5540  col1:=ord(rb); col2:=ord(rc)
5550 END;
5560
```

```
5570 (* scrsetflashing sets FOR how long each OF the two
5580   colours FOR the inks AND the border are TO be
5590   displayed on the screen. These settings apply TO
5600   all inks AND the border. The flash periods are given
5610   IN 1/50th OF a second ( 1/60th IN the USA ). The
5620   default is 10. *)
5630
5640 PROCEDURE scrsetflashing(p1,p2:integer);
5650 BEGIN
5660  rb:=chr(p1); rc:=chr(p2);
5670  user(#bc3e)
5680 END;
5690
5700 (* scrgetflashing returns the current flash periods as
5710   described above. *)
5720
5730 PROCEDURE scrgetflashing(VAR p1,p2:integer);
5740 BEGIN
5750  user(#bc41);
5760  p1:=ord(rb); p2:=ord(rc)
5770 END;
5780
5790 (* scrfillbox fills a character area OF the screen WITH an
5800   ink. col1 AND col2 are the left AND right columns OF the
5810   area TO be filled. row1 AND row2 are the top AND bottom
5820   rows OF the area. Coordinates are physical coordinates. *)
5830
5840 PROCEDURE scrfillbox(ink,col1,col2,row1,row2:integer);
5850 BEGIN
5860  ra:=chr(ink);
5870  rh:=chr(col1); rd:=chr(col2);
5880  rl:=chr(row1); re:=chr(row2);
5890  user(#bc44)
5900 END;
5910
5920 (* scrcharinvert inverts a character position. All pixels
5930   at the character postion that are written IN one ink
5940   are rewritten IN the other ink. The coordinates used
5950   are physical coordinates. *)
5960
5970 PROCEDURE scrcharinvert(i1,i2,col,row:integer);
5980 BEGIN
5990  rb:=chr(i1); rc:=chr(i2);
6000  rh:=chr(col); rl:=chr(row);
6010  user(#bc4a)
6020 END;
6030
```

```
6040 (* scrhwroll moves the whole screen up OR down 8 pixels
6050    (one character) using the hardware. The blank line is
6060    filled WITH the ink parameter colour. *)
6070
6080 PROCEDURE scrhwroll(up:boolean; ink:integer);
6090 BEGIN
6100  rb:=chr(ord(up)); ra:=chr(ink);
6110  user(#bc4d)
6120 END;
6130
6140 (* scrswroll moves an area OF the screen up OR down
6150    8 pixels (one character) using software. The area
6160    coordinates are given IN physical coordinates. *)
6170
6180 PROCEDURE scrswroll(up:boolean;ink,col1,col2,row1,row2:integer);
6190 BEGIN
6200  rb:=chr(ord(up)); ra:=chr(ink);
6210  rh:=chr(col1); rd:=chr(col2);
6220  rl:=chr(row1); re:=chr(row2);
6230  user(#bc50)
6240 END;
6250
6260 (* scraccess sets the screen write mode FOR the Graphics
6270    VDU. Possible mode values are
6280    0: FORCE  mode  NEW=INK
6290    1: XOR mode:     NEW= INK exculsive-OR OLD
6300    2: AND mode:     NEW= INK AND OLD
6310    3: OR mode:      NEW= INK OR OLD
6320
6330    NEW is the final setting OF the pixel.
6340    OLD is the current setting OF the pixel.
6350    INK is the ink plotted.
6360
6370    The default is FORCE mode (mode 0). *)
6380
6390 PROCEDURE scraccess(m:integer);
6400 BEGIN
6410  ra:=chr(m);
6420  user(#bc59)
6430 END;
6440
```

```
(* scrhorizontal plots a purely horizontal line using
  the current graphics write mode. *)

PROCEDURE scrhorizontal(ink,x1,x2,y:integer);
BEGIN
 ra:=chr(ink);
 rde:=x1; rbc:=x2; rhl:=y;
 user(#bc5f)
END;

(* scrvertical plots a purely vertical line using the
  current graphics write mode. *)

PROCEDURE scrvertical(ink,x,y1,y2:integer);
BEGIN
 ra:=chr(ink);
 rde:=x; rhl:=y1; rbc:=y2;
 user(#bc62)
END;

(* soundreset resets the Sound Manager; no more sounds are
  generated AND clear all queues. *)

PROCEDURE soundreset;
BEGIN
 user(#bca7)
END;

(* soundhold stops all sounds IN midflight. The result
  OF this FUNCTION is true IF a sound was active. *)

FUNCTION soundhold:boolean;
BEGIN
 user(#bcb6);
 soundhold:= odd(raf)
END;

(* soundcontinue restarts sounds after they have all been
  held. *)

PROCEDURE soundcontinue;
BEGIN
 user(#bcb9)
END;
```

```

(* soundaaddress returns the address OF the amplitude
   envelope given by its parameter. *)

FUNCTION soundaaddress(e:integer):integer;
BEGIN
 ra:=chr(e);
 user(#bcc2);
 soundaaddress:=rhl
END;

(* soundtaddress returns the address OF the tone envelope
   given by its parameter. *)

FUNCTION soundtaddress(e:integer):integer;
BEGIN
 ra:=chr(e);
 user(#bcc5);
 soundtaddress:=rhl
END;

(* mcprintchar tries TO send a character TO the Centronics
   port. The FUNCTION returns true IF it is successful;
   otherwise it returns false after about Ø.4 seconds. *)

FUNCTION mcprintchar(c:char):boolean;
BEGIN
 ra:=c;
 user(#bd2b);
 mcprintchar:=odd(raf)
END;

(* mcbusyprinter returns true IF the Centronics port
   is busy otherwise false is returned. *)

FUNCTION mcbusyprinter:boolean;
BEGIN
 user(#bd2e);
 mcbusyprinter:=odd(raf)
END;

```